The SGRUD Thesis

SGRUD is Growing Rapidly Until Distinction

Philip Schildkamp

Abstract

Abstract

Contents

List of Figures

List of Tables

Preformatted

```
*Text*
```

Table

row	row	row
col	col	col
col	col	col
col	col	col

References

Appendix

Module: bin

bin

• **bin**: `Object`

`@sgrud/bin` - The SGRUD CLI.

```
Description
  @sgrud/bin - The SGRUD CLI

Usage
  $ sgrud <command> [options]

Available Commands
  construct    Builds a SGRUD-based project using `microbundle`
  kickstart    Kickstarts a SGRUD-based project
  postbuild    Replicates exported package metadata for SGRUD-based projects
  runtimify    Creates ESM or UMD bundles for ES6 modules using `microbundle`
  universal    Runs SGRUD in universal (SSR) mode using `puppeteer`

For more info, run any command with the `--help` flag
  $ sgrud construct --help
  $ sgrud kickstart --help

Options
  -v, --version    Displays current version
  -h, --help       Displays this message
```

Defined in

packages/bin/index.ts:32

---

bin.construct

construct

▸ **construct**(`options?`): `Promise<void>`

Builds a SGRUD-based project using microbundle.

```
Description
  Builds a SGRUD-based project using `microbundle`

Usage
  $ sgrud construct [...modules] [options]

Options
  --compress    Compress/minify build output  (default true)
  --format      Build specified formats  (default commonjs,modern,umd)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud construct # Run with default options
```

```
  $ sgrud construct ./project/module # Build ./project/module
  $ sgrud construct ./module --format umd # Build ./module as umd
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.construct();
```

Example

Build `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./project/module']
});
```

Example

Build `./module` as `umd`:

```
require('@sgrud/bin');

sgrud.bin.construct({
  modules: ['./module'],
  format: 'umd'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.compress?`	`boolean`	Compress/minify build output. **`Default Value`** `true`
`options.format?`	`string`	Build specified formats. **`Default Value`** `'commonjs,modern,umd'`
`options.modules?`	`string[]`	Modules to build. **`Default Value`** `package.json#sgrud.construct`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/construct.ts:75

---

bin.kickstart

kickstart

▸ **kickstart**(`options?`): `Promise<void>`

Kickstarts a SGRUD-based project.

```
Description
  Kickstarts a SGRUD-based project

Usage
  $ sgrud kickstart [library] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
```

```
  $ sgrud kickstart # Run with default options
  $ sgrud kickstart preact --prefix ./module # Kickstart preact in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.kickstart();
```

Example

Kickstart `preact` in `./module`:

```
require('@sgrud/bin');

sgrud.bin.kickstart({
  prefix: './module',
  library: 'preact'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.library?`	`string`	Library which to base upon. **`Default Value`** `'sgrud'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/kickstart.ts:56

---

bin.postbuild

postbuild

▸ **postbuild**(`options?`): `Promise<void>`

Replicates exported package metadata for SGRUD-based projects.

```
Description
  Replicates exported package metadata for SGRUD-based projects

Usage
  $ sgrud postbuild [...modules] [options]

Options
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud postbuild # Run with default options
  $ sgrud postbuild ./project/module # Postbuild ./project/module
  $ sgrud postbuild --prefix ./module # Run in ./module
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.postbuild();
```

Example

Postbuild `./project/module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  modules: ['./project/module']
});
```

Example

Run in `./module`:

```
require('@sgrud/bin');

sgrud.bin.postbuild({
  prefix: './module'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.modules?`	`string[]`	Modules to build. **`Default Value`** `package.json#sgrud.postbuild`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/postbuild.ts:69

---

bin.runtimify

runtimify

▸ **runtimify**(`options?`): `Promise<void>`

Creates ESM or UMD bundles for ES6 modules using microbundle.

```
Description
  Creates ESM or UMD bundles for ES6 modules using `microbundle`

Usage
  $ sgrud runtimify [...modules] [options]

Options
  --format      Runtimify bundle format (umd or esm)  (default umd)
  --output      Output file in module root  (default runtimify.[format].js)
  --prefix      Use an alternative working directory  (default ./)
  -h, --help    Displays this message

Examples
  $ sgrud runtimify # Run with default options
  $ sgrud runtimify @microsoft/fast # Runtimify `@microsoft/fast`
```

Example

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.runtimify();
```

Example

Runtimify `@microsoft/fast`:

```
require('@sgrud/bin');

sgrud.bin.runtimify({
  modules: ['@microsoft/fast']
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.format?`	`string`	Runtimify bundle format (umd or esm). **`Default Value`** `'umd'`
`options.modules?`	`string[]`	Modules to runtimify. **`Default Value`** `package.json#sgrud.runtimify`
`options.output?`	`string`	Output file in module root. **`Default Value`** `'runtimify.[format].js'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/runtimify.ts:62

---

bin.universal

universal

▸ **universal**(`options?`): `Promise<void>`

Runs SGRUD in universal (SSR) mode using puppeteer.

```
Description
  Runs SGRUD in universal (SSR) mode using `puppeteer`

Usage
  $ sgrud universal [entry] [options]

Options
  --chrome      Chrome executable  (default /usr/bin/chromium-browser)
  --prefix      Use an alternative working directory  (default ./)
  -H, --host    Host to bind to  (default 127.0.0.1)
  -p, --port    Port to bind to  (default 4000)
  -h, --help    Displays this message

Examples
  $ sgrud universal # Run with default options
  $ sgrud universal --host 0.0.0.0 # Listen on all IPs
  $ sgrud universal -H 192.168.0.10 -p 4040 # Listen on 192.168.0.10:4040
```

`Example`

Run with default options:

```
require('@sgrud/bin');

sgrud.bin.universal();
```

`Example`

Listen on all IPs:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '0.0.0.0'
});
```

Example

Listen on `192.168.0.10:4040`:

```
require('@sgrud/bin');

sgrud.bin.universal({
  host: '192.168.0.10',
  port: '4040'
});
```

Parameters

Name	Type	Description
`options`	`Object`	Options object.
`options.chrome?`	`string`	Chrome executable. **`Default Value`** `'/usr/bin/chromium-browser'`
`options.entry?`	`string`	HTML document (relative to `prefix`). **`Default Value`** `'index.html'`
`options.host?`	`string`	Host to bind to. **`Default Value`** `'127.0.0.1'`
`options.port?`	`string`	Port to bind to. **`Default Value`** `'4000'`
`options.prefix?`	`string`	Use an alternative working directory. **`Default Value`** `'./'`

Returns `Promise<void>`

Execution promise.

Defined in packages/bin/src/universal.ts:74

Module: bus

bus

• **bus**: `Object`

`@sgrud/bus` - The SGRUD software bus.

The functionality implemented within this package is intended to ease the internal communication of applications building upon the `@sgrud` libraries. By establishing conduits between different modules of an application or between an application and plugins extending it, loose coupling of data transferral and functionality can be achieved.

Defined in packages/bus/index.ts:13

---

bus.BusHandle

BusHandle

T **BusHandle**: ‘$string$.{string}.${string}‘

The **BusHandle** is a string literal helper type which enforces any assigned string to contain at least three dots. It represents a domain name in reverse notation and thereby designates a hierarchical structure, which the *BusHandler* in conjunction with the *BusWorker* operate upon.

Example

Library-wide **BusHandle**:

```ts
import { BusHandle } from '@sgrud/bus';

const conduitHandle: BusHandle = 'io.github.sgrud';
```

Example

An invalid **BusHandle**:

```ts
import { BusHandle } from '@sgrud/bus';

const conduitHandle: BusHandle = 'org.example';
// Error: Type [...] is not assignable to type 'BusHandle'.
```

See

BusHandler

Defined in packages/bus/src/handler/handler.ts:30

---

bus.BusHandler

BusHandler

• **BusHandler**: `Object`

The **BusHandler** is a *Singleton* class, implementing and orchestrating the establishment, transferral and deconstruction of conduits in conjunction with the *BusWorker* process. To designate different conduits, the string literal helper type *BusHandle* is employed

As an example, let the following hierarchical structure be given:

```
io.github.sgrud
├── io.github.sgrud.core
│   ├── io.github.core.httpState
│   └── io.github.core.kernel
├── io.github.sgrud.data
│   └── io.github.data.model.current
│   ├── io.github.data.model.global
└── io.github.sgrud.shell
    ├── io.github.shell.catch
    └── io.github.shell.resolve
```

Decorator

Singleton

See

BusWorker

Defined in packages/bus/src/handler/handler.ts:100

---

bus.BusHandler.worker

worker

■ `Static` `Readonly` **worker**: `Thread<BusWorker>`

*Spawn*ed worker process and main conduit workhorse. The underlying *BusWorker* is run inside a `WebWorker` context and handles all published and subscribed conduits and the aggregation of their values depending on their hierarchy.

Decorator

Spawn

See

BusWorker

Defined in packages/bus/src/handler/handler.ts:112

---

bus.BusHandler.constructor

constructor

• **new BusHandler**(`tuples?`)

Public BusHandler constructor. As the BusHandler is a transparent Singleton, calling the `new` operator on it will always yield the same instance. The `new` operator can therefore be used to bulk-publish conduits.

`Example`

Set the `'io.github.sgrud.example'` conduit:

```
import { BusHandler } from '@sgrud/bus';
import { of } from 'rxjs';

new BusHandler([
  ['io.github.sgrud.example', of('published')]
]);
```

Parameters

Name	Type	Description
`tuples?`	[‘*string*.{string}.${string}‘, `Observable<any>`][]	List of conduits to publish.

Defined in packages/bus/src/handler/handler.ts:133

---

bus.BusHandler.get

get

▸ **get**<T>(handle): `Observable<BusValue<T>>`

Gets the conduit representing the supplied `handle`. Calling this method yields an Observable originating from the BusWorker which emits all BusValues published under the supplied `handle`. When getting `'io.github.sgrud'`, all conduits published hierarchically beneath this `handle`, e.g., `'io.github.bus.status'`, will also be emitted by the returned Observable.

`Example`

Get the `'io.github.sgrud'` conduit:

```
import { BusHandler } from '@sgrud/bus';

const conduitHandler = new BusHandler();
conduitHandler.get('io.github.sgrud').subscribe(console.log);
```

`See`

- BusHandle
- BusValue

Type parameters

Name	Description
T	Bus type.

Parameters

Name	Type	Description
`handle`	*'string*.{string}.${string}'	Bus handle.

Returns `Observable<BusValue<T»`

Observable.

Defined in packages/bus/src/handler/handler.ts:165

---

bus.BusHandler.set

set

▸ **set**<T>(handle, conduit): void

Publishes the supplied `conduit` under the supplied `handle`. Calling this method registers the supplied *Observable* with the *BusWorker*. When the *Observable* `conduit` completes, the registration will self-destruct. When overwriting a registration by supplying a previously used `handle` in conjunction with a different `conduit`, the previously supplied *Observable* will be unsubscribed.

Example

Set the `'io.github.sgrud.example'` conduit:

```
import { BusHandler } from '@sgrud/bus';
import { of } from 'rxjs';

const conduitHandler = new BusHandler();
conduitHandler.set('io.github.sgrud.example', of('published'));
```

See

- BusHandle
- BusValue

Type parameters

Name	Description
T	Bus type.

Parameters

Name	Type	Description
`handle`	*'string*.{string}.${string}'	Bus handle.
`conduit`	`Observable<T>`	Observable.

Returns `void`

Defined in packages/bus/src/handler/handler.ts:197

---

bus.BusValue

BusValue

• **BusValue**<T>: `Object`

The **BusValue** is an interface describing the shape of all values emitted by any active conduit. As conduits are *Observable* streams, which are dynamically merged through their hierarchical structure and therefore may emit more than one `value` from more than one `handle`, each value emitted by any conduit contains its originating `handle` and its typed internal `value`.

Example

```ts
import { BusHandler } from '@sgrud/bus';

const conduitHandler = new BusHandler();
conduitHandler.get('io.github.sgrud').subscribe(console.log);
// Result: { handle: 'io.github.sgrud.example', value: true }
```

See

BusHandler

Type parameters

Name	Description
`T`	Bus `value` type.

Defined in packages/bus/src/handler/handler.ts:52

---

bus.BusValue.handle

handle

• `Readonly` **handle**: ‘*string*.{string}.${string}‘

Emitting *BusHandle*.

See

BusHandle

Defined in packages/bus/src/handler/handler.ts:59

---

bus.BusValue.value

value

• `Readonly` **value**: `T`

Emitted value.

Defined in packages/bus/src/handler/handler.ts:64

---

bus.BusWorker

BusWorker

• **BusWorker**: `Object`

TODO

The *Web Worker spawn*ed by the *BusHandler* handling all published and subscribed conduits and the aggregation of their values depending on their hierarchy.

Decorator

Thread

Decorator

Singleton

See

- BusHandler
- Spawn

Defined in packages/bus/src/worker/index.ts:19

---

bus.BusWorker.constructor

constructor

• **new BusWorker**()

Public BusWorker constructor. This constructor is called once when Spawning the `WebWorker` running this class.

See

BusHandler

Defined in packages/bus/src/worker/index.ts:40

---

bus.BusWorker.get

get

▸ **get**(`handle`): `Observable<BusValue<any»`

Gets the conduit for the supplied `handle`. This method is called by the BusHandler and is only then proxied to the `WebWorker` running this class.

See

BusHandler

Parameters

Name	Type	Description
`handle`	‘*string*.{string}.${string}‘	Bus handle.

Returns `Observable<BusValue<any»`

Observable.

Defined in packages/bus/src/worker/index.ts:55

---

bus.BusWorker.set

set

▸ **set**(`handle`, `conduit`): `void`

Sets the supplied `conduit` for the supplied `handle`. This method is called by the BusHandler and is only then proxied to the `WebWorker` running this class.

See

BusHandler

Parameters

Name	Type	Description
`handle`	‘*string*.{string}.${string}‘	Bus handle.
`conduit`	`Observable<any>`	Observable.

Returns `void`

Defined in packages/bus/src/worker/index.ts:79

---

bus.BusWorker.changes

changes

• `Private Readonly` **changes**: `BehaviorSubject<BusWorker>`

BehaviorSubject emitting every time a conduit is added or deleted from the internal conduits map. This emittance is used to recompile the open subscriptions previously obtained to through use of the get method.

Defined in packages/bus/src/worker/index.ts:26

---

bus.BusWorker.conduits

conduits

• `Private Readonly` **conduits**: `Map<`‘$string$.{string}.${string}‘, `Observable<BusValue<any»>`

Internal map containing all established conduits. Updating this map should always be accompanied by an emittance of the changes.

Defined in packages/bus/src/worker/index.ts:32

---

bus.Publish

Publish

▸ **Publish**(`handle`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. This decorator publishes the decorated property value under the supplied `handle`. If the supplied `source` is a string it is assumed to reference a property key of the prototype containing the decorated property. The first instance value assigned to this `source` property is assigned as readonly on the instance and appended to the supplied `handle`, thus creating an *instance-scoped handle*. This *scoped handle* is then used to publish the first instance value assigned to the decorated property. This implies that the publication will wait until both the decorated property and the referenced `source` property are assigned values. If the supplied `source` is of an Observable type, this Observable is published under the supplied `handle` and assigned as readonly to the decorated prototype property. If no `source` is supplied, a new Subject will be created and implicitly supplied as `source`.

Precautions should be taken to ensure completion of the supplied Observable source as otherwise memory leaks may occur due to dangling subscriptions.

`Example`

Publish the `'io.github.sgrud.example'` conduit:

```
import { Publish } from '@sgrud/bus';
import type { Subject } from 'rxjs';

export class Publisher {

  @Publish('io.github.sgrud.example')
  public readonly conduit!: Subject<any>;

}

Publisher.prototype.conduit.complete();
```

`Example`

Publish the `'io.github.sgrud.example'` conduit:

```
import { Publish } from '@sgrud/bus';
import { Subject } from 'rxjs';

export class Publisher {
```

```
  @Publish('io.github.sgrud', 'scope')
  public readonly conduit: Subject<any> = new Subject<any>();

  public constructor(
    private readonly scope: string
  ) { }

}

const publisher = new Publisher('example');
publisher.conduit.complete();
```

See

- BusHandler
- Subscribe

Parameters

Name	Type	Description
`handle`	*`string`*.{string}.${string}`	Bus handle.
`source`	`string` | `Observable<any>`	Property key or Observable.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/publish.ts:67

---

bus.Subscribe

Subscribe

▸ **Subscribe**(`handle`, `source?`): (`prototype`: `object`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. This decorator assigns an Observable emitting BusValues originating from the supplied `handle` to the decorated property. If no `source` is supplied, this Observable is assigned as readonly to the decorated prototype property. Else the supplied `source` is assumed to be a string referencing a property key of the prototype containing the decorated property. The first instance value assigned to this `source` property is assigned as readonly on the instance and appended to the supplied `handle`, thus creating an *instance-scoped handle*. This *scoped handle* is then used to create an Observable which is assigned as readonly to the decorated property on the instance. This implies that the decorated property will not be assigned an Observable until the referenced `source` property is assigned an instance value.

Example

Subscribe to the `'io.github.sgrud.example'` conduit:

```
import type { BusValue } from '@sgrud/bus';
import { Subscribe } from '@sgrud/bus';
import type { Observable } from 'rxjs';

export class Subscriber {
```

```
  @Subscribe('io.github.sgrud.example')
  public readonly conduit!: Observable<BusValue<any>>;

}
```

Example

Subscribe to the `'io.github.sgrud.example'` conduit:

```
import type { BusValue } from '@sgrud/bus';
import { Subscribe } from '@sgrud/bus';
import type { Observable } from 'rxjs';

export class Subscriber {

  @Subscribe('io.github.sgrud', 'scope')
  public readonly conduit!: Observable<BusValue<any>>;

  public constructor(
    public readonly scope: string
  ) { }

}

const subscriber = new Subscriber('example');
```

See

- BusHandler
- Publish

Parameters

Name	Type	Description
`handle`	*'string.{string}.${string}'*	Bus handle.
`source?`	`string`	Property key.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/bus/src/handler/subscribe.ts:61

Module: core

core

• **core**: `Object`

`@sgrud/core` - The SGRUD core module.

The functions and classes found within this module represent the base upon which the SGRUD client library is built. Therefore, most of the code provided within this module does not aim to fulfill one specific high-level need, but is intended to be used as low-level building block for downstream projects. This practice is employed throughout the SGRUD client library, as all modules depend on this core module. By

providing the core functionality within this singular module, all downstream SGRUD modules should be considered opt-in functionality which may be used within projects building upon the SGRUD client library.

Defined in packages/core/index.ts:17

---

core.Assign

Assign

T **Assign**<S, T>: { [K in keyof (S & T)]: K extends keyof S ? S[K] : K extends keyof T ? T[K] : never }

Type helper assigning the own property types of all of the enumerable own properties from a source type to a target type.

Example

Assign `valueOf()` to `string`:

```
import type { Assign } from '@sgrud/core';

const str = 'Hello world' as Assign<{
  valueOf(): 'Hello world';
}, string>;
```

Type parameters

Name	Description
S	Source type.
T	Target type.

Defined in packages/core/src/typing/assign.ts:18

---

core.Factor

Factor

▸ **Factor**<K>(targetFactory): (prototype: object, propertyKey: PropertyKey) => void

Prototype property decorator factory. Replaces the decorated prototype property with a getter, which looks up the linked instance of a target constructor forwarded-referenced by the `linkFactory()`.

Example

Factor a service:

```
import { Factor } from '@sgrud/core';
import { Service } from './service';

export class ServiceHandler {

  @Factor(() => Service)
  private readonly service!: Service;

}
```

See

- Linker
- Target

Type parameters

Name	Type	Description
K	extends () => any	Target constructor type.

Parameters

Name	Type	Description
`targetFactory`	() => `K`	Forward reference to the target constructor.

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`object`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/linker/factor.ts:29

---

core.HttpClient

HttpClient

• **HttpClient**: `Object`

The HttpClient is a thin wrapper around the `rxjs/ajax` method. The main function of this wrapper is to pipe all requests through a chain of classes extending HttpProxy. Thereby interceptors for various requests can be implemented to, e.g., provide API credentials etc.

`Decorator`

Singleton

Defined in packages/core/src/http/client.ts:38

---

core.HttpClient.delete

delete

▸ `Static` **delete**<`T`>(`url`): `Observable<AjaxResponse<T>`

Fires a `HTTP DELETE` request upon subscription. Shorthand for calling handle with respective arguments.

`Example`

Fire a `DELETE` request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.delete('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T»`

Observable response.

Defined in packages/core/src/http/client.ts:56

---

core.HttpClient.get

get

▸ `Static` **get**<T>(`url`): `Observable<AjaxResponse<T»`

Fires a `HTTP GET` request upon subscription. Shorthand for calling handle with respective arguments.

`Example`

Fire a `GET` request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.get('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T»`

Observable response.

Defined in packages/core/src/http/client.ts:76

---

core.HttpClient.head

head

▸ `Static` **head**<T>(`url`): `Observable<AjaxResponse<T»`

Fires a `HTTP HEAD` request upon subscription. Shorthand for calling handle with respective arguments.

`Example`

Fire a `HEAD` request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.head('https://example.com').subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.

Returns `Observable<AjaxResponse<T»`

Observable response.

Defined in packages/core/src/http/client.ts:96

---

core.HttpClient.patch

patch

▸ `Static` **patch**<T>(`url`, `body`): `Observable<AjaxResponse<T»`

Fires a `HTTP PATCH` request upon subscription. Shorthand for calling handle with respective arguments.

`Example`

Fire a `PATCH` request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.patch('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`url`	`string`	Request URL.
`body`	`unknown`	Request body.

Returns `Observable<AjaxResponse<T»`

Observable response.

Defined in packages/core/src/http/client.ts:119

---

core.HttpClient.post

post

▸ Static **post**<T>(url, body): Observable<AjaxResponse<T>»

Fires a HTTP POST request upon subscription. Shorthand for calling handle with respective arguments.

Example

Fire a POST request against https://example.com:

```
import { HttpClient } from '@sgrud/core';

HttpClient.post('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
url	string	Request URL.
body	unknown	Request body.

Returns Observable<AjaxResponse<T>»

Observable response.

Defined in packages/core/src/http/client.ts:142

---

core.HttpClient.put

put

▸ Static **put**<T>(url, body): Observable<AjaxResponse<T>»

Fires a HTTP PUT request upon subscription. Shorthand for calling handle with respective arguments.

Example

Fire a PUT request against https://example.com:

```
import { HttpClient } from '@sgrud/core';

HttpClient.put('https://example.com', {
  bodyContent: 'value'
}).subscribe(console.log);
```

Type parameters

Name	Description
T	Response type.

Name	Type	Description

Parameters

Name	Type	Description
`url`	`string`	Request URL.
`body`	`unknown`	Request body.

Returns `Observable<AjaxResponse<T»`

Observable response.

Defined in packages/core/src/http/client.ts:165

---

core.HttpClient.constructor

constructor

• **new HttpClient**()

---

core.HttpClient.handle

handle

▸ **handle**<T>(`request`): `Observable<AjaxResponse<T»`

Generic handle method, enforced by the HttpHandler interface. Main method of the HttpClient. Internally pipes the request config through all linked classes extending HttpProxy.

`Example`

Fire a custom request against `https://example.com`:

```
import { HttpClient } from '@sgrud/core';

HttpClient.prototype.handle({
  method: 'GET',
  url: 'https://example.com',
  headers: { 'x-example': 'value' }
}).subscribe(console.log);
```

`See`

HttpProxy

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Request.

Returns `Observable<AjaxResponse<T»`

Observable response.

Implementation of HttpHandler.handle

Defined in packages/core/src/http/client.ts:192

---

core.HttpHandler

HttpHandler

• **HttpHandler**: `Object`

The HttpHandler interface enforces the generic handle method with `rxjs/ajax` compliant typing on the implementing class or object. Used by HttpProxy to type the next hops in the proxy chain.

`See`

- HttpClient
- HttpProxy

Defined in packages/core/src/http/client.ts:15

---

core.HttpHandler.handle

handle

▸ **handle**<T>(`request`): `Observable<AjaxResponse<T>`

Generic method enforcing `rxjs/ajax` compliant typing. The method signature corresponds to that of the `rxjs/ajax` method itself.

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Request.

Returns `Observable<AjaxResponse<T>`

Observable response.

Defined in packages/core/src/http/client.ts:25

---

core.HttpProxy

HttpProxy

• `Abstract` **HttpProxy**: `Object`

Abstract base class to implement proxies, i.e., HTTP request interceptors, on the client side. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the respective classes proxy method will be called with the request details (which could have been modified by a previous HttpProxy) and the next HttpHandler (which could be the next HttpProxy or the `rxjs/ajax` method), when a request is fired through the HttpClient.

`Example`

Simple proxy intercepting `file:` requests:

```ts
import type { HttpHandler, HttpProxy } from '@sgrud/core';
import { Provider, Target } from '@sgrud/core';
import { Observable, of } from 'rxjs';
import type { AjaxConfig, AjaxResponse } from 'rxjs/ajax';
import { file } from './file';

@Target<typeof FileProxy>()
export class FileProxy
  extends Provider<typeof HttpProxy>('sgrud.core.http.HttpProxy') {

  public override proxy<T>(
    request: AjaxConfig,
    handler: HttpHandler
  ): Observable<AjaxResponse<T>> {
    if (request.url.startsWith('file:')) {
      return of<AjaxResponse<T>>(file);
    }

    return handler.handle<T>(request);
  }
}
```

Decorator

Provide

See

HttpClient

Defined in packages/core/src/http/proxy.ts:45

---

core.HttpProxy.[provide]

[provide]

■ `Static Readonly` **[provide]**: `"sgrud.core.http.HttpProxy"`

Magic string by which this class is provided.

See

provide

Defined in packages/core/src/http/proxy.ts:52

---

core.HttpProxy.constructor

constructor

• **new HttpProxy**()

---

core.HttpProxy.proxy

proxy

▸ `Abstract` **proxy**<T>(`request`, `handler`): `Observable<AjaxResponse<T>`

The overridden proxy method of linked classes extending HttpProxy is called whenever a request is fired through the HttpClient. The extending class can either pass the `request` to the next `handler`, with or without modifying it, or an interceptor can chose to completely handle a `request` by itself through returning an Observable.

Type parameters

Name	Description
T	Response type.

Parameters

Name	Type	Description
request	AjaxConfig	Request.
handler	HttpHandler	Next handler.

Returns `Observable<AjaxResponse<T>`

Observable response.

Defined in packages/core/src/http/proxy.ts:67

---

core.HttpState

HttpState

• **HttpState**: `Object`

Built-in HttpProxy intercepting all requests fired through the HttpClient. This proxy implements [observable], through which it emits an array of all currently open connections every time a new request is fired or a running request is completed.

`Decorator`

Target

`Decorator`

Singleton

`See`

- HttpClient
- HttpProxy

Defined in packages/core/src/http/state.ts:22

---

core.HttpState.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.core.http.HttpProxy"`

Magic string by which this class is provided.

`See`

provide

Inherited from HttpProxy.[provide]

Defined in packages/core/src/http/proxy.ts:52

---

core.HttpState.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<AjaxResponse<any>[]>`

Type declaration ▸ (): `Subscribable<AjaxResponse<any>[]>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits an array of all active requests whenever this list mutates. Using the returned Subscribable, e.g., a load indicator can easily be implemented.

`Example`

Subscribe to the currently active requests:

```
import { HttpState, Linker } from '@sgrud/core';
import { from } from 'rxjs';

const httpState = new Linker<typeof HttpState>().get(HttpState);
from(httpState).subscribe(console.log);
```

Returns `Subscribable<AjaxResponse<any>[]>`

Callback to a Subscribable.

Defined in packages/core/src/http/state.ts:43

---

core.HttpState.constructor

constructor

• **new HttpState**()

Public constructor. Called by the Target decorator to link this HttpProxy into the proxy chain.

Overrides HttpProxy.constructor

Defined in packages/core/src/http/state.ts:70

---

core.HttpState.proxy

proxy

▸ **proxy**<T>(`request`, `handler`): `Observable<AjaxResponse<T>`

Overridden proxy method of the HttpProxy base class. Mutates the request to also emit progress events while the request is running. These progress events will be consumed by the HttpState interceptor and re-supplied via the Observable returned by the [observable] getter.

Type parameters

Name	Description
`T`	Response type.

Parameters

Name	Type	Description
`request`	`AjaxConfig`	Request.
`handler`	`HttpHandler`	Next handler.

Returns `Observable<AjaxResponse<T>`

Observable response.

Overrides HttpProxy.proxy

Defined in packages/core/src/http/state.ts:89

---

core.HttpState.changes

changes

• `Private Readonly` **changes**: `BehaviorSubject<HttpState>`

BehaviorSubject emitting every time a request is added to or deleted from the internal running map.

Defined in packages/core/src/http/state.ts:49

---

core.HttpState.running

running

• `Private Readonly` **running**: `Map<AjaxConfig, AjaxResponse<any»`

Internal map containing all running requests. Updating this map should always be accompanied by an emittance of the changes.

Defined in packages/core/src/http/state.ts:55

---

core.Kernel

Kernel

• **Kernel**: `Object`

Singleton Kernel class. This Kernel is essentially a dependency loader for ESM bundles (and their respective importmap) or, depending on the browser context, UMD bundles and their transitive dependencies. Using this Kernel, applications based on the SGRUD client library may be comprised of multiple, optionally loaded Modules, which, depending on the application structure and configuration, can be insmodded initially, by supplying them as sgrudDependencies through the corresponding API endpoint, or later on, manually.

`Decorator`

Singleton

Defined in packages/core/src/kernel/kernel.ts:14

packages/core/src/kernel/kernel.ts:139

---

core.Kernel.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<Module>`

Type declaration ▸ (): `Subscribable<Module>`

Symbol property typed as callback to a Subscribable. The returned subscribable stream emits all the loaded modules (and never completes).

`Example`

Subscribe to the stream of loaded modules:

```
import { Kernel } from '@sgrud/core';
import { from } from 'rxjs';

from(new Kernel()).subscribe(console.log);
```

Returns `Subscribable<Module>`

Callback to a Subscribable.

Defined in packages/core/src/kernel/kernel.ts:156

---

core.Kernel.constructor

constructor

• **new Kernel**(`baseHref?`, `endpoint?`, `nodeModules?`)

Singleton Kernel constructor. The first time, the Kernel is instantiated, it will retrieve the list of modules which should be loaded and then insmods all those modules and their transitive dependencies. Every subsequent constructor call will ignore all arguments and return the singleton instance. Through subscribing to the Subscribable returned by the `rxjs.observable` interop getter, the initial loading progress can be tracked.

`Example`

Instantiate the Kernel:

```
import { Kernel } from '@sgrud/core';

const kernel = new Kernel(
  'https://example.com',
  '/context/api/sgrud',
  'https://unpkg.com'
);
```

Parameters

Name	Type	Default value	Description
`baseHref`	`string`	`location.origin`	Base href for building URLs.
`endpoint`	`string`	`undefined`	Href of the SGRUD API endpoint.
`nodeModules`	`string`	`undefined`	Href to load node modules from.

Defined in packages/core/src/kernel/kernel.ts:222

---

core.Kernel.endpoint

endpoint

• `Readonly` **endpoint**: `string`

The global SGRUD API endpoint. The Kernel will, e.g., request the list of modules to be loaded (by their names) from `${endpoint}/insmod`.

Defined in packages/core/src/kernel/kernel.ts:233

---

core.Kernel.insmod

insmod

▸ **insmod**(`module`, `source?`, `entryModule?`): `Observable<Module>`

Insert modules. Calling this method while supplying a valid `module` definition will chain the `module` dependencies and the `module` itself into an Observable, which is then returned. When multiple `modules` are inserted, their dependencies are deduplicated by internally tracking all `modules` and their transitive dependencies as separate loaders. Depending on the browser context, either the UMD or ESM bundles (and their respective importmap) are loaded via calling script. When inserting Module.sgrudDependencies,

their compatibility is checked. Should a dependency version mismatch, the returned Observable will throw a RangeError.

Example

Insert a module by definition:

```
import { Kernel } from '@sgrud/core';
import packageJson from 'module/package.json';

new Kernel().insmod(packageJson).subscribe(console.log);
```

See

Module

Parameters

Name	Type	Default value	Description
`module`	`Module`	`undefined`	Module definition.
`source`	`string`	`undefined`	Optional module source.
`entryModule`	`boolean`	`false`	Wether to run the module.

Returns `Observable<Module>`

Observable of the module loading.

Defined in packages/core/src/kernel/kernel.ts:292

---

core.Kernel.nodeModules

nodeModules

• `Readonly` **nodeModules**: `string`

The global path to load Nodejs modules from. All JavaScript assets belonging to modules installed via NPM should be located here.

Defined in packages/core/src/kernel/kernel.ts:239

---

core.Kernel.resolve

resolve

▸ **resolve**(`name`, `source?`): `Observable<Module>`

Retrieves a module definition by module `name`. The module `name` is appended to the node module path and the package.json file therein retrieved via HTTP GET. The parsed package.json is then emitted by the returned Observable.

Example

Resolve a module definition by name:

```
import { Kernel } from '@sgrud/core';

new Kernel().resolve('module').subscribe(console.log);
```

See

Module

Parameters

Name	Type	Description
`name`	`string`	Module name.
`source`	`string`	Optional module source.

Returns `Observable<Module>`

Observable of the module definition.

Defined in packages/core/src/kernel/kernel.ts:419

---

core.Kernel.script

script

▸ **script**(`props`): `Observable<void>`

Inserts a HTML script element and applies the supplied `props` to it. The returned Observable emits and completes when the element's onload handler is called. When no external resource is supplied through `props.src`, the onload handler is asynchronously called. When the returned Observable completes, the inserted HTML script element is removed.

`Example`

Insert a HTML script element:

```
import { Kernel } from '@sgrud/core';

new Kernel().script({
  src: '/node_modules/module/bundle.js',
  type: 'text/javascript'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLScriptElement>`	Script element properties.

Returns `Observable<void>`

Observable of the script loading.

Defined in packages/core/src/kernel/kernel.ts:453

---

core.Kernel.verify

verify

▸ **verify**(`props`): `Observable<void>`

Inserts a HTML link element and applies the supplied `props` to it. This method should be used to verify a module before importing and evaluating it, by providing its subresource integrity.

`Example`

Insert a HTML link element:

```
import { Kernel } from '@sgrud/core';

new Kernel().verify({
  href: '/node_modules/module/index.js',
  integrity: 'sha256-[...]',
  rel: 'modulepreload'
}).subscribe();
```

Parameters

Name	Type	Description
`props`	`Partial<HTMLLinkElement>`	Link element properties.

Returns `Observable<void>`

Deferred link appendage and removal.

Defined in packages/core/src/kernel/kernel.ts:494

---

core.Kernel.imports

imports

• `Private` `Readonly` **imports**: `Map<string, string>`

Internal mapping of all via importmap defined module identifiers to their corresponding paths. This mapping is used for housekeeping, e.g., to prevent the same module identifier to be defined multiple times.

Defined in packages/core/src/kernel/kernel.ts:163

---

core.Kernel.loaders

loaders

• `Private` `Readonly` **loaders**: `Map<string, ReplaySubject<Module»`

Internal mapping of all insmodded modules to a corresponding ReplaySubject. The ReplaySubject tracks the module loading process as such, that it emits the module definition once the respective module is fully loaded (including dependencies etc.) and then completes.

Defined in packages/core/src/kernel/kernel.ts:171

---

core.Kernel.loading

loading

• `Private` `Readonly` **loading**: `ReplaySubject<Module>`

Internal ReplaySubject tracking the loading state of the Kernel modules. An Observable from this ReplaySubject may be retrieved by subscribing to the Subscribable returned by the `rxjs.observable` interop getter. The actual ReplaySubject (and therefore Observable) emits all module definitions which are insmodded throughout the lifespan of the Kernel.

Defined in packages/core/src/kernel/kernel.ts:180

---

core.Kernel.shimmed

shimmed

• `Private` `Readonly` **shimmed**: `string`

Internally used string to suffix the `importmap` and `module` types of HTML script elements with, if applicable. This string is set to whatever trails the type of HTML script elements encountered upon initialization, iff their type starts with `importmap`.

Defined in packages/core/src/kernel/kernel.ts:188

---

core.Kernel

Kernel

• **Kernel**: `Object`

Namespace containing types and interfaces to be used in conjunction with the singleton Kernel class.

See

Kernel

Defined in packages/core/src/kernel/kernel.ts:14

packages/core/src/kernel/kernel.ts:139

---

core.Kernel.Digest

Digest

T **Digest**: ‘sha256‖384‖512−{string}‘

String literal helper type. Enforces any assigned string to adhere to the represent a browser-parsable digest hash.

See

core.Module

Defined in packages/core/src/kernel/kernel.ts:22

---

core.Kernel.Module

Module

• **Module**: `Object`

Interface describing the shape of a module. This interface is aligned with some package.json fields. It further specifies an optional ependencies mapping, as well as an optional webDependencies mapping, which both are used by the Kernel to determine SGRUD module dependencies and runtime (web) dependencies.

Example

An example module definition:

```
import type { Kernel } from '@sgrud/core';

const module: Kernel.Module = {
  name: 'module',
  version: '0.0.0',
  exports: './module.exports.js',
  unpkg: './module.unpkg.js',
  sgrudDependencies: {
    sgrudDependency: '^0.0.1'
  },
  webDependencies: {
    webDependency: {
      exports: {
        webDependency: './webDependency.exports.js'
      },
      unpkg: [
        './webDependency.unpkg.js'
      ]
    }
  }
};
```

See

Kernel

Defined in packages/core/src/kernel/kernel.ts:60

Kernel.Module.digest

digest

• `Optional` `Readonly` **digest**: `Record<string, 'sha256-`$string || sha384 - \{string\}$`' | 'sha512-${string}'>`

Optional bundle digests. If hashes are supplied, they will be used to verify the subresource integrity of the respective bundles.

See

Digest

Defined in packages/core/src/kernel/kernel.ts:89

Kernel.Module.exports

exports

• `Optional` `Readonly` **exports**: `string`

ESM module entry point.

Defined in packages/core/src/kernel/kernel.ts:75

Kernel.Module.name

name

• `Readonly` **name**: `string`

Name of the module.

Defined in packages/core/src/kernel/kernel.ts:65

Kernel.Module.sgrudDependencies

sgrudDependencies

• `Optional` `Readonly` **sgrudDependencies**: `Record<string, string>`

Optional SGRUD dependencies.

Defined in packages/core/src/kernel/kernel.ts:94

Kernel.Module.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string`

UMD module entry point.

Defined in packages/core/src/kernel/kernel.ts:80

Kernel.Module.version

version

• `Readonly` **version**: `string`

Module version, formatted as semver.

Defined in packages/core/src/kernel/kernel.ts:70

Kernel.Module.webDependencies

webDependencies

• `Optional` `Readonly` **webDependencies**: `Record<string, WebDependency>`

Optional runtime dependencies.

See

WebDependency

Defined in packages/core/src/kernel/kernel.ts:101

---

core.Kernel.WebDependency

WebDependency

• **WebDependency**: `Object`

Interface describing a runtime dependency of a Module.

See

Module

Defined in packages/core/src/kernel/kernel.ts:110

Kernel.WebDependency.exports

exports

• `Optional` `Readonly` **exports**: `Record<string, string>`

Optional ESM runtime dependencies.

Defined in packages/core/src/kernel/kernel.ts:115

Kernel.WebDependency.unpkg

unpkg

• `Optional` `Readonly` **unpkg**: `string[]`

Optional UMD runtime dependencies.

Defined in packages/core/src/kernel/kernel.ts:120

---

core.Linker

Linker

• **Linker**<K, V>: `Object`

Linker is the Singleton link map used by the Factor decorator to lookup the linked instances of targeted constructors. To programmatically insert some links, the inherited `MapConstructor` or `Map.prototype.set` methods are available. The former will insert all entries into this singleton link map, internally calling the latter for each.

`Example`

Preemptively link an instance:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>([[
  Service,
  new Service('linked')
]]);
```

`Decorator`

Singleton

`See`

- Factor
- Target

Type parameters

Name	Type	Description
K	extends () => V	Target constructor type.
V	InstanceType<K>	Linked instance type.

Defined in

packages/core/src/linker/linker.ts:38

---

core.Linker.constructor

constructor

• **new Linker**<K, V>(`entries?`)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
entries?	null | readonly readonly [K, V][]

Inherited from

Map<K, V>.constructor

Defined in

node_modules/typescript/lib/lib.es2015.collection.d.ts:53

• **new Linker**<K, V>(`iterable?`)

Type parameters

Name	Type
K	extends () => V
V	InstanceType<K>

Parameters

Name	Type
iterable?	null | Iterable<readonly [K, V]>

Inherited from Map<K, V>.constructor

Defined in node_modules/typescript/lib/lib.es2015.iterable.d.ts:161

---

core.Linker.get

get

▸ **get**(target): V

Overridden Map.prototype.get method. Looks up the linked instance based on the target constructor. If no linked instance is found, one is created by calling the new operator on the target constructor. Therefor the target constructors must not require parameters (i.e. all parameters have to be optional).

Example

Retrieve a linked instance:

```
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().get(Service);
```

Parameters

Name	Type	Description
target	K	Target constructor.

Returns V

Linked instance.

Overrides Map.get

Defined in packages/core/src/linker/linker.ts:62

---

core.Linker.getAll

getAll

▸ **getAll**(target): V[]

Returns all linked instances, which satisfy instanceof target. Use this method when multiple linked constructors extend the same base class and are to be retrieved.

Example

Retrieve all linked instances:

```ts
import { Linker } from '@sgrud/core';
import { Service } from './service';

new Linker<typeof Service>().getAll(Service);
```

Parameters

Name	Type	Description
`target`	`K`	Target constructor.

Returns `V[]`

Linked instances.

Defined in packages/core/src/linker/linker.ts:90

---

core.Merge

Merge

T **Merge**<T>: `T extends T ? (_: T) => T : never extends (_: infer I) => T ? I : never`

Type helper to convert union types (`A | B`) to intersection types (`A & B`).

Remarks

https://github.com/microsoft/TypeScript/issues/29594

Type parameters

Name	Description
`T`	Union type.

Defined in packages/core/src/typing/merge.ts:8

---

core.Mutable

Mutable

T **Mutable**<T>: { -readonly [K in keyof T]: T[K] }

Type helper marking the supplied type as mutable (opposed to readonly).

Remarks

https://github.com/Microsoft/TypeScript/issues/24509

Type parameters

Name	Type	Description
`T`	extends `object`	Type to be marked mutable.

Defined in packages/core/src/typing/mutable.ts:8

---

core.Provide

Provide

`T` **Provide**`<K, V>: (...args: any[]) => InstanceType<V> & { [provide]: K extends Registration ? K : Registration }`

Type helper enforcing the provide symbol property containing a magic string (typed as Provider) on base constructors decorated with the corresponding `@Provide()` decorator.

See

Provider

Type parameters

Name	Type	Description
`K`	extends `Registration`	Magic string type.
`V`	extends `(...args: any[]) => InstanceType<V>`	Providing constructor type.

Defined in packages/core/src/super/provide.ts:68

packages/core/src/super/provide.ts:22

---

core.Provide

Provide

▸ **Provide**`<V, K>(): (constructor: V) => void`

Class decorator factory. Provides the decorated constructor by magic string to extending classes. Applying this decorator enforces the corresponding `Provide` type and thereby the provider constraint on the decorated class, i.e., constructor. This contract enforces the declaration of a (static) provide symbol property typed as Registration. The magic string value of this static property is used by the Provider to lookup base constructors within the Registry map.

Example

Provide a base class:

```
import { Provide, provide } from '@sgrud/core';

@Provide<typeof Base>()
export abstract class Base {

  public static readonly [provide]:
  'sgrud.example.Base' = 'sgrud.example.Base' as const;

}
```

See

- Provider
- Registry

Type parameters

Name	Type	Description
`V`	extends `Provide<K, V>`	Providing constructor type.
`K`	extends ‘$string.{string}.${string}‘ = V[typeof provide]	Magic string type.

Returns `fn`

Class decorator.

▸ `(constructor): void`

Parameters

Name	Type
`constructor`	`V`

Returns `void`

Defined in packages/core/src/super/provide.ts:68

---

core.Provider

Provider

▸ **Provider**<`V`, `K`>(`provider`): `V`

Provider of base classes. Extending this mixin-style function while supplying the `typeof` a Provided constructor enforces type safety and hinting on the supplied magic string and the resulting class which `extends` this Provider mixin. The main purpose of this pattern is bridging module gaps by de-coupling bundle files while maintaining a well-defined prototype chain. This still requires the base class to be defined (and Provided) before extension but allows intellisense'd OOP patterns across multiple modules while maintaining runtime language specifications.

`Example`

Extend a provided class:

```
import { Provider } from '@sgrud/core';
import type { Base } from 'example-module';

export class Class
  extends Provider<typeof Base>('org.example.Base') {

  public constructor(...args: any[]) {
    super(...args);
  }

}
```

`See`

- Provide
- Registry

Type parameters

Name	Type	Description
`V`	extends `Provide<K, V>`	Providing constructor type.
`K`	extends ‘$string.{string}.${string}‘ = `V[typeof provide]`	Magic string type.

Parameters

Name	Type	Description
`provider`	`K`	Magic string.

Returns `V`

Providing constructor.

Defined in packages/core/src/super/provider.ts:65

---

core.Provider

Provider

• **Provider**<V>: `Object`

Type helper to allow referencing Provided constructors as `new`-able targets. Use in conjunction with the `@Provide()` decorator.

See

Provide

Type parameters

Name	Description
V	Providing instance type.

Defined in packages/core/src/super/provider.ts:65

packages/core/src/super/provider.ts:12

---

core.Provider.[provide]

[provide]

• `Readonly` **[provide]**: ‘*string*.{string}.${string}‘

Enforced provider contract.

See

provide

Defined in packages/core/src/super/provider.ts:19

---

core.Provider.constructor

constructor

• **constructor**: `Object`

---

core.Provider.constructor

constructor

• **new Provider**(...args)

Enforced constructor contract.

Parameters

Name	Type	Description
...args	any[]	Class constructor rest parameter.

Defined in packages/core/src/super/provider.ts:27

---

core.Registration

Registration

T **Registration**: *'string.{string}.${string}'*

String literal helper type. Enforces any assigned string to contain at least three dots. Registrations are used to lookup constructors by magic strings through classes extending the base Provider and should represent sane package paths in dot-notation.

Example

Library-wide Registration pattern:

```
import type { Registration } from '@sgrud/core';

const registration: Registration = 'sgrud.module.path.ClassName';
```

See

- Provide
- Provider
- Registry

Defined in packages/core/src/super/registry.ts:21

---

core.Registry

Registry

• **Registry**<K, V>: `Object`

The Registry is a Singleton map used by the Provider to lookup Provided constructors by magic strings upon class extension. Magic strings should represent sane package paths in dot-notation. To programmatically provide constructors by magic strings to extending classes, the inherited `MapConstructor` or `Map.prototype.set` methods are available. The former will insert all entries into this singleton Registry map, internally calling the latter for each. Whenever a currently no registered constructor is requested, an intermediary class is created, cached internally and returned. When the actual constructor is registered, the previously created intermediary class is removed from the internal caching and further steps are taken, to guarantee the transparent addressing of the actual constructor through the intermediary class.

Decorator

Singleton

Example

Preemptively provide a constructor by magic string:

```
import type { Registration } from '@sgrud/core';
import { Registry } from '@sgrud/core';
import { Service } from './service';

new Registry<Registration, typeof Service>([
  ['sgrud.example.Service', Service]
]);
```

See

- Provide
- Provider

Type parameters

Name	Type	Description
K	extends `Registration`	Magic string type.

Name	Type	Description
`V`	extends (...`args`: `any`[]) => `InstanceType`<`V`>	Providing constructor type.

Defined in packages/core/src/super/registry.ts:65

---

core.Registry.constructor

constructor

• **new Registry**<`K`, `V`>(`tuples?`)

Overridden `MapConstructor`. The constructor of this class accepts the same parameters as the overridden `MapConstructor` and acts the same. I.e., through instantiating this Registry singleton and passing a list of tuples of Registrations and their corresponding constructors, these tuples will be stored.

Type parameters

Name	Type
`K`	extends ‘$string$.{string}.${string}‘
`V`	extends (...`args`: `any`[]) => `InstanceType`<`V`>

Parameters

Name	Type	Description
`tuples?`	[`K`, `V`][]	List of constructors to provide.

Overrides Map<K, V>.constructor

Defined in packages/core/src/super/registry.ts:98

---

core.Registry.get

get

▸ **get**(`registration`): `V`

Overridden `Map.prototype.get` method. Looks up the Provided constructor by magic string. If no provided constructor is found, an intermediary class is created, cached internally and returned. While this intermediary class and the functionality supporting it takes care of inheritance, i.e., allows to forward-reference base classes to be extended, it cannot substitute for the actual extended constructor. Therefore, static extension of forward-referenced classes may be used, but as long as the actual extended constructor is not registered (and therefore the intermediary class is still acting as inheritance cache), the extending class cannot be instantiated, called etc. Doing so will result in a ReferenceError being thrown.

`Example`

Retrieve a provided constructor by magic string:

```
import type { Registration } from '@sgrud/core';
import { Registry } from '@sgrud/core';
import type { Service } from 'example-module';

new Registry<Registration, typeof Service>().get('org.example.Service');
```

Parameters

Name	Type	Description
`registration`	`K`	Magic string.

Returns `V`

Providing constructor.

Overrides Map.get

Defined in packages/core/src/super/registry.ts:137

---

core.Registry.set

set

▸ **set**(`registration`, `constructor`): `Registry<K, V>`

Overridden `Map.prototype.set` method. Whenever a constructor is provided by magic string through calling this method, a check is run, wether this constructor was previously requested and therefore was cached as intermediary class. If so, the intermediary class is removed from this internal map and further steps are taken, to guarantee the transparent addressing of the newly provided constructor through the previously cached intermediary class.

`Example`

Preemptively provide a constructor by magic string:

```ts
import type { Registration } from '@sgrud/core';
import { Registry } from '@sgrud/core';
import { Service } from './service';

new Registry<Registration, typeof Service>().set(
  'org.example.Service',
  Service
);
```

Parameters

Name	Type	Description
`registration`	`K`	Magic string.
`constructor`	`V`	Providing constructor.

Returns `Registry<K, V>`

This registry instance.

Overrides Map.set

Defined in packages/core/src/super/registry.ts:193

---

core.Registry.cached

cached

• `Private` `Readonly` **cached**: `Map<K, V>`

Internally used map of all cached, i.e., forward referenced, constructors. Whenever a constructor, which is not currently registered, is requested as a provider, an intermediary class is created and stored within this map until the actual constructor is registered. As soon as this happens, the intermediary class is removed from this map and further steps are taken, to guarantee the transparent addressing of the actual constructor through the intermediary class.

Defined in packages/core/src/super/registry.ts:79

---

core.Registry.caches

caches

• `Private Readonly` **caches**: `WeakSet<V>`

Internally used (weak) set containing all intermediary classes created upon requesting a currently not registered constructor as provider. This (weak) set is used internally to check, if a intermediary class has already been replaced by the actual constructor.

Defined in packages/core/src/super/registry.ts:87

---

core.Singleton

Singleton

▸ **Singleton**<T>(`apply?`): (`constructor`: T) => T

Class decorator factory. Enforces transparent singleton pattern on decorated class. When calling the `new` operator on a decorated class, if provided, the `apply` callback is fired with the singleton instance and the construction invocation parameters.

`Example`

Singleton class:

```
import { Singleton } from '@sgrud/core';

@Singleton<typeof Service>()
export class Service { }
```

Type parameters

Name	Type	Description
T	extends (...args: any[]) => any	Constructor type.

Parameters

Name	Type	Description
apply?	(self: InstanceType<T>, args: ConstructorParameters<T>) => InstanceType<T>	Construct function.

Returns `fn`

Class decorator.

▸ (`constructor`): T

Parameters

Name	Type
constructor	T

Returns T

Defined in packages/core/src/utility/singleton.ts:20

---

core.Spawn

Spawn

▸ **Spawn**(`worker`, `source?`): (`constructor`: (...`args`: `any`[]) => `any`, `propertyKey`: `PropertyKey`) => `void`

TODO

This class property decorator factory spawns a *Web Worker*, wraps it with *comlink* and assigns it to the decorated class property.

`Example`

Spawn a WebWorker:

```
import { Spawn, Thread } from '@sgrud/core';
import { WebWorker } from 'web-worker';

export class WebWorkerHandler {

  @Spawn('web-worker')
  private static readonly worker: Thread<WebWorker>;

}
```

`See`

- Thread
- comlink
- Web Worker

Parameters

Name	Type	Description
`worker`	`string` | `Endpoint` | `NodeEndpoint`	Worker constructor.
`source?`	`string`	Optional module source.

Returns `fn`

Class property decorator.

▸ (`constructor`, `propertyKey`): `void`

Parameters

Name	Type
`constructor`	(...`args`: `any`[]) => `any`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/core/src/thread/spawn.ts:38

---

core.Target

Target

▸ **Target**<K>(`factoryArgs?`, `target?`): (`constructor`: K) => `void`

Class decorator factory. Links the decorated target constructor to its corresponding instance by applying the decorated constructor arguments. Employ this helper to link target constructors with required arguments. Supplying a `target` constructor overrides it with the constructed instance.

Example

Target a service:

```
import { Target } from '@sgrud/core';

@Target<typeof Service>(['default'])
export class Service {

  public constructor(
    public readonly param: string
  ) { }

}
```

Example

Factor a targeted service:

```
import { Factor } from '@sgrud/core';
import type { Target } from '@sgrud/core';
import { Service } from './service';

export class ServiceHandler {

  @Factor<Target<Service>>(() => Service)
  public readonly service!: Service;

}
```

See

- Factor
- Linker

Type parameters

Name	Type	Description
K	extends (...args: any[]) => any	Target constructor type.

Parameters

Name	Type	Description
`factoryArgs?`	`ConstructorParameters<K>`	Arguments for the target constructor.
`target?`	K	Target constructor override.

Returns `fn`

Class decorator.

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	K

Returns `void`

Defined in packages/core/src/linker/target.ts:67

---

core.Target

Target

• **Target**<V>: `Object`

Type helper to allow Factoring targeted constructors with required arguments. Use in conjunction with the `@Target()` decorator.

See

Factor

Type parameters

Name	Description
V	Linked instance type.

Defined in packages/core/src/linker/target.ts:67

packages/core/src/linker/target.ts:11

---

core.Target.constructor

constructor

• **constructor**: `Object`

---

core.Target.constructor

constructor

• **new Target**(...args)

Enforced constructor contract.

Parameters

Name	Type	Description
...args	any[]	Class constructor rest parameter.

Defined in packages/core/src/linker/target.ts:19

---

core.Thread

Thread

T **Thread**<T>: `Promise<Remote<T>>`

TODO

Interface describing an exposed class in a remote context. Created by wrapping comlink.Remote in a Promise. Use in conjunction with the `@Thread()` decorator.

See

Thread

Type parameters

Name	Description
T	Thread instance type.

Defined in packages/core/src/thread/thread.ts:38

packages/core/src/thread/thread.ts:17

---

core.Thread

Thread

▸ **Thread**(): (`constructor`: () => `any`) => `void`

TODO

Class decorator factory. Exposes an instance of the decorated class as worker via comlink.expose.

Example

WebWorker thread:

```
import { Thread } from '@sgrud/core';

@Thread()
export class WebWorker { }
```

See

Spawn

Returns `fn`

Class decorator.

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	() => `any`

Returns `void`

Defined in packages/core/src/thread/thread.ts:38

---

core.TypeOf

TypeOf

• `Abstract` **TypeOf**: `Object`

Strict type-assertion and runtime type-checking utility. When type-checking variables in the global scope, e.g., `window` or `process`, make use of the `globalThis` object.

Example

Type-check global context:

```
import { TypeOf } from '@sgrud/core';

TypeOf.process(globalThis.process); // running in node context
TypeOf.window(globalThis.window);   // running in browser context
```

Defined in packages/core/src/utility/type-of.ts:15

---

core.TypeOf.array

array

▸ `Static` **array**(`value`): value is any[]

Type-check for `Array<any>`.

`Example`

Type-check `null` for `Array<any>`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.array(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is any[]

Whether `value` is of type `Array<any>`.

Defined in packages/core/src/utility/type-of.ts:31

---

core.TypeOf.boolean

boolean

▸ `Static` **boolean**(`value`): value is boolean

Type-check for `boolean`.

`Example`

Type-check `null` for `boolean`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.boolean(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is boolean

Whether `value` is of type `boolean`.

Defined in packages/core/src/utility/type-of.ts:49

---

core.TypeOf.date

date

▸ `Static` **date**(`value`): value is Date

Type-check for `Date`.

`Example`

Type-check `null` for `Date`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.date(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Date

Whether `value` is of type `Date`.

Defined in packages/core/src/utility/type-of.ts:67

---

core.TypeOf.function

function

▸ `Static` **function**(`value`): value is Function

Type-check for `Function`.

`Example`

Type-check `null` for `Function`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.function(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Function

Whether `value` is of type `Function`.

Defined in packages/core/src/utility/type-of.ts:85

---

core.TypeOf.global

global

▸ `Static` **global**(`value`): value is typeof globalThis

Type-check for `global`.

`Example`

Type-check `null` for `typeof globalThis`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.global(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is typeof globalThis

Whether `value` is of type `typeof globalThis`.

Defined in packages/core/src/utility/type-of.ts:103

---

core.TypeOf.null

null

▸ `Static` **null**(`value`): value is null

Type-check for `null`.

`Example`

Type-check `null` for `null`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.null(null); // true
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is null

Whether `value` is of type `null`.

Defined in packages/core/src/utility/type-of.ts:121

---

core.TypeOf.number

number

▸ `Static` **number**(`value`): value is number

Type-check for `number`.

`Example`

Type-check `null` for `number`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.number(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is number

Whether `value` is of type `number`.

Defined in packages/core/src/utility/type-of.ts:139

---

core.TypeOf.object

object

▸ `Static` **object**(`value`): value is object

Type-check for `object`.

`Example`

Type-check `null` for `object`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.object(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is object

Whether `value` is of type `object`.

Defined in packages/core/src/utility/type-of.ts:157

---

core.TypeOf.process

process

▸ `Static` **process**(`value`): value is Process

Type-check for `NodeJS.Process`.

`Example`

Type-check `null` for `NodeJS.Process`:

```
import { TypeOf } from '@sgrud/core';

TypeOf.process(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Process

Whether `value` is of type `NodeJS.Process`.

Defined in packages/core/src/utility/type-of.ts:175

---

core.TypeOf.promise

promise

▸ `Static` **promise**(`value`): value is Promise<any>

Type-check for `Promise<any>`.

`Example`

Type-check `null` for `Promise<any>`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.promise(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Promise<any>

Whether `value` is of type `Promise<any>`.

Defined in packages/core/src/utility/type-of.ts:193

---

core.TypeOf.regex

regex

▸ `Static` **regex**(`value`): value is RegExp

Type-check for `RegExp`.

`Example`

Type-check `null` for `RegExp`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.regex(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is RegExp

Whether `value` is of type `RegExp`.

Defined in packages/core/src/utility/type-of.ts:211

---

core.TypeOf.string

string

▸ `Static` **string**(`value`): value is string

Type-check for `string`.

`Example`

Type-check `null` for `string`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.string(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is string

Whether `value` is of type `string`.

Defined in packages/core/src/utility/type-of.ts:229

---

core.TypeOf.undefined

undefined

▸ `Static` **undefined**(`value`): value is undefined

Type-check for `undefined`.

`Example`

Type-check `null` for `undefined`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.undefined(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is undefined

Whether `value` is of type `undefined`.

Defined in packages/core/src/utility/type-of.ts:247

---

core.TypeOf.url

url

▸ `Static` **url**(`value`): value is URL

Type-check for `URL`.

`Example`

Type-check `null` for `URL`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.url(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is URL

Whether `value` is of type `URL`.

Defined in packages/core/src/utility/type-of.ts:265

---

core.TypeOf.window

window

▸ `Static` **window**(`value`): value is Window

Type-check for `Window`.

`Example`

Type-check `null` for `Window`:

```ts
import { TypeOf } from '@sgrud/core';

TypeOf.window(null); // false
```

Parameters

Name	Type	Description
`value`	`unknown`	Value to type-check.

Returns value is Window

Whether `value` is of type `Window`.

Defined in packages/core/src/utility/type-of.ts:283

---

core.TypeOf.constructor

constructor

• `Private` **new TypeOf**()

Private abstract constructor (which should never be called).

`Throws`

TypeError.

Defined in packages/core/src/utility/type-of.ts:292

---

core.assign

assign

▸ **assign**<T, S>(target, ...sources): T & Merge<S[number]>

Deep copy the values of all of the enumerable own properties from one or more source objects to a target object. Returns the target object.

Example

Deep copy nested properties:

```
import { assign } from '@sgrud/core';

assign(
  { one: { one: true }, two: false },
  { one: { key: null } },
  { two: true }
);

// { one: { one: true, key: null }, two: true },
```

Type parameters

Name	Type	Description
T	extends Record<PropertyKey, any>	Target type.
S	extends Record<PropertyKey, any>[]	Source types.

Parameters

Name	Type	Description
target	T	Target object to deep copy properties to.
...sources	[...S[]]	Source objects from which to deep copy properties.

Returns T & Merge<S[number]>

Target object.

Defined in packages/core/src/utility/assign.ts:28

---

core.pluralize

pluralize

▸ **pluralize**(singular): string

Pluralizes words of the English language.

Example

Pluralize 'money':

```
import { pluralize } from '@sgrud/core';

pluralize('money'); // 'money'
```

Example

Pluralize 'thesis':

```
import { pluralize } from '@sgrud/core';

pluralize('thesis'); // 'theses'
```

Parameters

Name	Type	Description
`singular`	`string`	English word in singular form.

Returns `string`

Plural form of `singular`.

Defined in packages/core/src/utility/pluralize.ts:23

---

core.provide

provide

• `Const` **provide**: typeof `provide`

Symbol used as property key by the Provide decorator to enforce the provider contract.

`See`

- Provide
- Provider

Defined in packages/core/src/super/provide.ts:10

---

core.semver

semver

▸ **semver**(`version`, `range`): `boolean`

Best-effort semver matcher. The supplied `version` will be tested against all supplied `range`.

`Example`

Test `'1.2.3'` against `'>2 <1 || ~1.2.*'`:

```ts
import { semver } from '@sgrud/core';

semver('1.2.3', '>2 <1 || ~1.2.*'); // true
```

Parameters

Name	Type	Description
`version`	`string`	Tested semantic version string.
`range`	`string`	Range to test the `version` against.

Returns `boolean`

Wether `version` satisfies `range`.

Defined in packages/core/src/kernel/semver.ts:17

Module: data

data

• **data**: `Object`

`@sgrud/data` - The SGRUD data model.

The functionality implemented within this package is intended to ease the type safe data handling, i.e., retrieval, mutation and storage, throughout applications building upon the `@sgrud` libraries. By extending the Model class and applying adequate decorators to the contained properties, the resulting extension will, in its static context, provide all necessary means to interact directly with the underlying repository, while the instance context of any class extending the abstract model base class will inherit methods to observe changes to its instance field values, selectively complement the instance with fields from the backing data storage via type safe graph representations and to delete the respective instance from the data storage.

Defined in packages/data/index.ts:19

---

data.Enum

Enum

• `Abstract` **Enum**: `Object`

Abstract Enum helper class. This class is used by the unravel method to detect enumerations within a Graph, as enumerations (in contrast to plain strings) must not be quoted. This class should never be instantiated manually, but instead is used internally by the enumerate function.

`See`

enumerate

Defined in packages/data/src/model/enum.ts:10

---

data.Enum.constructor

constructor

• `Private` **new Enum**()

Private abstract constructor (which should never be called).

`Throws`

TypeError.

Overrides String.constructor

Defined in packages/data/src/model/enum.ts:18

---

data.HasMany

HasMany

▸ **HasMany**<`T`>(`typeFactory`, `transient?`): <`M`>(`model`: `M`, `field`: `Field`<`M`>) => `void`

Model field decorator factory. Using this decorator, Models can be enriched with one-to-many associations to other Models. Any argument for the `typeFactory` has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument) be recognized when serializing or treemapping the Model containing the decorated field.

`Example`

Model with a has many association:

```ts
import { HasMany, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasMany(() => OwnedModel)
  public field?: OwnedModel[];

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- Property

Type parameters

Name	Type	Description
`T`	extends `Type<any, T>`	Field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	`() => T`	`undefined`	Forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is transient.

Returns `fn`

Model field decorator.

▸ `<M>(model, field): void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/has-many.ts:45

---

data.HasOne

HasOne

▸ **HasOne**<T>(typeFactory, transient?): <M>(model: M, field: Field<M>) => void

Model field decorator factory. Using this decorator, Models can be enriched with one-to-one associations to other Models. Any argument for the `typeFactory` has to be another Model. By applying this decorator, the decorated field will (depending on the `transient` argument) be recognized when serializing or treemapping the Model containing the decorated field.

Example

Model with a has one association:

```
import { HasOne, Model } from '@sgrud/data';
import { OwnedModel } from './owned-model';

export class ExampleModel extends Model<ExampleModel> {

  @HasOne(() => OwnedModel)
  public field?: OwnedModel;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasMany
- Property

Type parameters

Name	Type	Description
T	extends Type<any, T>	Field value constructor type.

Parameters

Name	Type	Default value	Description
typeFactory	() => T	undefined	Forward reference to the field value constructor.
transient	boolean	false	Whether the decorated field is transient.

Returns fn

Model field decorator.

▸ <M>(model, field): void

Type parameters

Name	Type
M	extends Model<any, M>

Parameters

Name	Type
model	M
field	Field<M>

Returns void

Defined in packages/data/src/relation/has-one.ts:45

---

data.HttpQuerier

HttpQuerier

• **HttpQuerier**: `Object`

HTTP based data querier, i.e., extension of the abstract Querier base class, allowing data queries to be committed via HTTP. To use this class, provide it to the Linker by either extending it, and decorating the extending class with the Target decorator, or by preemptively supplying an instance of this class to the Linker.

`Example`

Provide the HttpQuerier class to the Linker:

```
import { Linker } from '@sgrud/core';
import { HttpQuerier } from '@sgrud/data';

new Linker<typeof HttpQuerier>([[
  HttpQuerier,
  new HttpQuerier('https://api.example.com')
]]);
```

`See`

- Model
- Querier

Defined in packages/data/src/querier/http.ts:28

---

data.HttpQuerier.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.querier.Querier"`

Magic string by which this class is provided.

`See`

provide

Inherited from Querier.[provide]

Defined in packages/data/src/querier/querier.ts:90

---

data.HttpQuerier.commit

commit

▸ **commit**(`operation`, `variables`): `Observable<any>`

Overridden `commit` method of the Querier base class. When this Model querier is made available via the Linker, this overridden method is called whenever this querier claims the highest priority to commit an operation, depending on the Model from which the Operation originates.

Parameters

Name	Type	Description
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Querier operation to be committed.
`variables`	`Variables`	Variables within the operation.

Returns `Observable<any>`

Observable of the committed operation.

Overrides Querier.commit

Defined in packages/data/src/querier/http.ts:68

---

data.HttpQuerier.constructor

constructor

• **new HttpQuerier**(`endpoint?`, `prioritize?`)

Public constructor consuming the HTTP `endpoint` Model queries should be committed against, and an dynamic or static `prioritize` value. The `prioritize` value may either be a mapping of Models to corresponding priorities or a static priority for this querier.

Parameters

Name	Type	Default value	Description
`endpoint`	`string`	`undefined`	HTTP querier endpoint.
`prioritize`	`number | Map<Type<any>, number>`	`0`	Dynamic or static prioritization.

Overrides Querier.constructor

Defined in packages/data/src/querier/http.ts:50

---

data.HttpQuerier.priority

priority

▸ **priority**(`model`): `number`

Overridden priority method of the Querier base class. When a Operation is to be committed, this method is called with the respective `model` constructor and returns the claimed priority to commit this Model.

Parameters

Name	Type	Description
`model`	`Type<any>`	Model to be committed.

Returns `number`

Priority of this implementation.

Overrides Querier.priority

Defined in packages/data/src/querier/http.ts:91

---

data.HttpQuerier.types

types

• `Readonly` **types**: `Set<Type>`

A set containing the the Types this Model querier can handle. As HTTP connections are short-lived, this querier may only handle one-off querier types, namely `'mutation'` and `'query'`.

Overrides Querier.types

Defined in packages/data/src/querier/http.ts:36

---

data.HttpQuerier.endpoint

endpoint

• `Private` `Readonly` **endpoint**: `string`

HTTP querier endpoint.

Defined in packages/data/src/querier/http.ts:51

---

data.HttpQuerier.prioritize

prioritize

• `Private` `Readonly` **prioritize**: `number` | `Map<Type<any>, number>` = `0`

Dynamic or static prioritization.

Defined in packages/data/src/querier/http.ts:52

---

data.Model

Model

• `Abstract` **Model**<`M`>: `Object`

Abstract base class to implement data models. By extending this abstract base class while providing the enforced symbol property containing the singular name of the resulting data model, type safe data handling, i.e., retrieval, mutation and storage, can easily be achieved. Through the use of the static- and instance-scoped polymorphic `this`, all inherited operations warrant type safety and provide intellisense.

`Example`

Extend the model base class:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field: string?;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

Type parameters

Name	Type	Description
`M`	extends `Model` = `any`	Extending model instance type.

Defined in packages/data/src/model/model.ts:18

packages/data/src/model/model.ts:119

packages/data/src/model/model.ts:318

---

data.Model.commit

commit

▸ `Static` **commit**<T>(`this`, `operation`, `variables?`): `Observable<any>`

Static commit method. Calling this method on a class extending the abstract model base class, while supplying an `operation` and all its embedded `variables`, will dispatch the supplied operation to the respective model repository through the highest priority Querier or, if no querier is compatible, throw an error. This method is the central point of origin for all model-related data transferral and is internally called by all other distinct methods of the model.

`Throws`

Observable of ReferenceError.

`Example`

Commit a `query`-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

`See`

Querier

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
operation	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Operation to commit.
variables	Variables	Variables within the operation.

Returns `Observable<any>`

Observable of the commitment.

Defined in packages/data/src/model/model.ts:352

---

data.Model.deleteAll

deleteAll

▸ `Static` **deleteAll**<`T`>(`this`, `uuids`): `Observable<any>`

Static deleteAll method. Calling this method on a class extending the model, while supplying a list of `uuids`, will dispatch the deletion of all model instances identified by these UUIDs to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-deletions from the respective model repository can be achieved.

`Example`

Delete all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`uuids`	`string[]`	UUIDs of model instances to be deleted.

Returns `Observable<any>`

Observable of the deletion.

Defined in packages/data/src/model/model.ts:403

---

data.Model.deleteOne

deleteOne

▸ `Static` **deleteOne**<`T`>(`this`, `uuid`): `Observable<any>`

Static deleteOne method. Calling this method on a class extending the model, while supplying an `uuid`, will dispatch the deletion of the model instance identified by this UUID to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the deletion of a single model instance from the respective model repository can be achieved.

`Example`

Delete one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`uuid`	`string`	UUID of the model instance to be deleted.

Returns `Observable<any>`

Observable of the deletion.

Defined in packages/data/src/model/model.ts:437

---

data.Model.findAll

findAll

▸ `Static` **findAll**<T>(`this`, `filter`, `graph`): `Observable<{ result: T[] ; total: number }>`

Static findAll method. Calling this method on a class extending the abstract model base class, while supplying a `filter` to match model instances by and a `graph` containing the fields to be included in the result, will dispatch the lookup operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the bulk-lookup of model instances from the respective model repository can be achieved.

`Example`

Find all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-01-01')
          }
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'id',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
filter	Filter<T>	Filter to find model instances by.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<{ result: T[] ; total: number }>`

Observable of the find operation.

Defined in packages/data/src/model/model.ts:496

---

data.Model.findOne

findOne

▸ Static **findOne**<T>(this, shape, graph): Observable<T>

Static findOne method. Calling this method on a class extending the abstract model base class, while supplying the shape to match the model instance by and a graph describing the fields to be included in the result, will dispatch the lookup operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the retrieval of one specific model instance from the respective model repository can be achieved.

Example

Find one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
shape	Shape<T>	Shape of the model instance to find.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<T>`

Observable of the find operation.

Defined in packages/data/src/model/model.ts:544

---

data.Model.saveAll

saveAll

▸ `Static` **saveAll**<T>(`this`, `models`, `graph`): `Observable<T[]>`

Static saveAll method. Calling this method on a class extending the abstract model base class, while supplying a list of `models` which to save and a `graph` describing the fields to be included in the result, will dispatch the save operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-persistance of model instances from the respective model repository can be achieved.

`Example`

Persist multiple models:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
models	T[]	Array of models to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<T[]>`

Observable of the save operation.

Defined in packages/data/src/model/model.ts:590

---

data.Model.saveOne

saveOne

▸ `Static` **saveOne**<T>(`this`, `model`, `graph`): `Observable<T>`

Static saveOne method. Calling this method on a class extending the abstract model base class, while supplying a `model` which to save and a `graph` describing the fields to be included in the result, will dispatch the save operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, persistance of one specific model instance from the respective model repository can be achieved.

`Example`

Persist a model:

```ts
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
model	T	Model which is to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<T>`

Observable of the save operation.

Defined in packages/data/src/model/model.ts:632

---

data.Model.serialize

serialize

▸ Static **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static serialize method. Calling this method on a class extending the model, while supplying a `model` which to serialize and optionally enabling `shallow` serialization, will return the shape of the model, i.e., a plain JSON representation of all model fields, or undefined, if the supplied `model` does not contain any fields or values. By serializing `shallow`ly, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific model instance from the respective model repository can be achieved.

Example

Serialize a model:

```ts
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	Static polymorphic this.
model	T	undefined	Model which is to be serialized.

Name	Type	Default value	Description
`shallow`	`boolean`	`false`	Whether to serialize shallowly.

Returns `undefined | Shape<T>`

Shape of the model or undefined.

Defined in packages/data/src/model/model.ts:674

---

data.Model.treemap

treemap

▸ `Static` **treemap**<T>(this, model, shallow?): `undefined | Graph<T>`

Static treemap method. Calling this method on a class extending the abstract model base class, while supplying a `model` which to treemap and optionally enabling `shallow` treemapping, will return a graph describing the fields which are declared and defined on the supplied `model`, or undefined, if the supplied `model` does not contain any fields or values. By treemapping `shallow`ly, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the graph for one specific model instance from the respective model repository can be retrieved.

`Example`

Treemap a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Default value	Description
`this`	`Type<T>`	`undefined`	Static polymorphic this.
`model`	`T`	`undefined`	Model which is to be treemapped.
`shallow`	`boolean`	`false`	Whether to treemap shallowly.

Returns `undefined | Graph<T>`

Graph of the model or undefined.

Defined in packages/data/src/model/model.ts:742

---

data.Model.unravel

unravel

▸ Static **unravel**<T>(this, graph): string

Static unravel method. Calling this method on a class extending the abstract model base class, while supplying a graph describing the fields which to unravel, will return the unraveled graph as raw string. Through this method, the graph for one specific model instance from the respective model repository can be unraveled into a raw string. This unraveled graph can than be consumed by, e.g., the commit method.

Example

Unravel a graph:

```
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'id',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
graph	Graph<T>	Graph which is to be unraveled.

Returns string

Unraveled graph as raw string.

Defined in packages/data/src/model/model.ts:806

---

data.Model.valuate

valuate

▸ Static **valuate**<T>(this, model, field): any

Static valuate method. Calling this method on a class extending the abstract model base class, while supplying a model and a field which to valuate, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied field of the supplied model. Through this method, the preprocessed field value of one specific model instance from the respective model repository can be retrieved.

Example

Valuate a field:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`model`	`T`	Model which is to be valuated.
`field`	`Field<T>`	Field of the model to be valuated.

Returns `any`

Valuated field value.

Defined in packages/data/src/model/model.ts:875

---

data.Model.[hasMany]

[hasMany]

• `Optional` `Readonly` **[hasMany]**: `Record<keyof M, () => unknown>`

Symbol property used by the HasMany decorator.

See

HasMany

Defined in packages/data/src/model/model.ts:933

---

data.Model.[hasOne]

[hasOne]

• `Optional` `Readonly` **[hasOne]**: `Record<keyof M, () => unknown>`

Symbol property used by the HasOne decorator.

See

HasOne

Defined in packages/data/src/model/model.ts:926

---

data.Model.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<M>`

Type declaration ▸ (): `Subscribable<M>`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits every mutation this model instance experiences.

Example

Subscribe to a model instance:

```ts
import { from } from 'rxjs';
import { ExampleModel } from './example-model';

const model = new ExampleModel();
from(model).subscribe(console.log);
```

Returns `Subscribable<M>`

Callback to a Subscribable.

Defined in packages/data/src/model/model.ts:958

---

data.Model.[property]

[property]

• `Optional` `Readonly` **[property]**: `Record<keyof M, () => unknown>`

Symbol property used by the Property decorator.

See

Property

Defined in packages/data/src/model/model.ts:940

---

data.Model.assign

assign

▸ **assign**<T>(this, ...parts): `Observable<T>`

Instance-scoped assign method. Calling this method, while supplying a list of `parts`, will assign all supplied `parts` to the model instance. The assignment is implemented as deep merge assignment. Using this method, an existing model instance can easily be mutated while still emitting the mutated changes.

Example

Assign parts to a model instance:

```ts
import { ExampleModel } from './example-model';

const model = new ExampleModel();
model.assign({ field: 'example' }).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends `Model<any, T> = M`	Extending model instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.
...parts	Shape<T>[]	Array of parts to assign.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1061

---

data.Model.clear

clear

▸ **clear**<T>(this, keys?): Observable<T>

Instance-scoped clear method. Calling this method on an instance of a class extending the abstract model base class, while optionally supplying a list of keys which are to be cleared, will set the value of the properties described by either the supplied keys or, if no keys were supplied, all enumerable properties of the class extending the abstract model base class to undefined, effectively clearing them.

Example

Clear a model instance selectively:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
model.clear(['field']).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.
keys?	Field<T>[]	Optional array of keys to clear.

Returns Observable<T>

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1092

---

data.Model.commit

commit

▸ **commit**<T>(this, operation, variables?, mapping?): Observable<T>

Instance-scoped commit method. Internally calls commit on the this-context of an instance of a class extending the abstract model base class and furthermore assigns the returned data to the model instance the find method was called upon. When supplying a mapping, the returned data will be mutated by the supplied OperatorFunction (otherwise this mapping defaults to rxjs.identity).

Example

Commit a query-type operation:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel();

model.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

See

Querier

Type parameters

Name	Type	Description
`T`	extends `Model<any, T> = M`	Extending model instance type.

Parameters

Name	Type	Default value	Description
`this`	`T`	`undefined`	Polymorphic this.
`operation`	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	`undefined`	GraphQL Operation to commit.
`variables`	`Variables`	`{}`	Variables within the `operation`.
`mapping`	`OperatorFunction<any, Shape<T>>`	`identity`	Mapping to apply to the result.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1149

---

data.Model.constructor

constructor

• **new Model**<`M`>(...`parts`)

Public constructor. The constructor of all classes extending the abstract model base class, unless explicitly overridden, behaves analogous to the instance-scoped assign method, as it takes all supplied `parts` and assigns them to the instantiated and returned model. The constructor furthermore wires some internal functionality, e.g., creates a new changes BehaviorSubject which emits every mutation this model instance experiences.

Type parameters

Name	Type
`M`	extends `Model<any, M> = any`

Parameters

Name	Type	Description
`...parts`	`Shape<M>[]`	Array of parts to assign.

Defined in packages/data/src/model/model.ts:1034

---

data.Model.created

created

• `Optional` **created**: `Date`

Transient creation date of this model instance.

Defined in packages/data/src/model/model.ts:970

---

data.Model.delete

delete

▸ **delete**<T>(this): Observable<T>

Instance-scoped delete method. Internally calls deleteOne while supplying the UUID of an instance of a class extending the abstract model base class. Calling this method furthermore clears the model instance and completes its deletion by calling complete on the internal changes BehaviorSubject of the model instance the delete method was called upon.

Example

Delete a model instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});

model.delete().subscribe(console.log);
```

See

deleteOne

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.

Returns Observable<T>

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1187

---

data.Model.find

find

▸ **find**<T>(this, graph, shape?): Observable<T>

Instance-scoped find method. Internally calls findOne on the this-context of an instance of a class extending the abstract model base class and furthermore assigns the returned data to the model instance the find method was called upon.

Example

Find a model instance by UUID:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({
  id: '3068b30e-82cd-44c5-8912-db13724816fd'
});
```

```
model.find([
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

See

findOne

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.
graph	Graph<T>	Graph of fields to be included.
shape	Shape<T>	Shape of the model to find.

Returns Observable<T>

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1226

---

data.Model.id

id

• Optional **id**: string

Universally unique identifier of this model instance.

Defined in packages/data/src/model/model.ts:964

---

data.Model.modified

modified

• Optional **modified**: Date

Transient modification date of this model instance.

Defined in packages/data/src/model/model.ts:976

---

data.Model.save

save

▸ **save**<T>(this, graph?): Observable<T>

Instance-scoped save method. Internally calls saveOne on the this-context of an instance of a class extending the abstract model base class and furthermore assigns the returned data to the model instance the save method was called upon.

Example

Persist a model instance:

```ts
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });

model.save([
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

See

saveOne

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Description
this	T	Polymorphic this.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<T>`

Observable of the mutated instance.

Defined in packages/data/src/model/model.ts:1263

---

data.Model.serialize

serialize

▸ **serialize**<T>(this, shallow?): undefined | Shape<T>

Instance-scoped serialize method. Internally calls serialize on the this-context of an instance of a class extending the abstract model base class.

Example

Serialize a model instance:

```ts
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.serialize()); // { field: 'example' }
```

See

serialize

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Default value	Description
this	T	undefined	Polymorphic this.
shallow	boolean	false	Whether to serialize shallowly.

Returns `undefined | Shape<T>`

Shape of this instance or undefined.

Defined in packages/data/src/model/model.ts:1293

---

data.Model.treemap

treemap

▸ **treemap**<T>(this, shallow?): undefined | Graph<T>

Instance-scoped treemap method. Internally calls treemap on the this-context of an instance of a class extending the abstract model base class.

Example

Treemap a model instance:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
console.log(model.treemap()); // ['field']
```

See

treemap

Type parameters

Name	Type	Description
T	extends Model<any, T> = M	Extending model instance type.

Parameters

Name	Type	Default value	Description
this	T	undefined	Polymorphic this.
shallow	boolean	false	Whether to treemap shallowly.

Returns `undefined | Graph<T>`

Graph of this instance or undefined.

Defined in packages/data/src/model/model.ts:1321

---

data.Model.[toStringTag]

[toStringTag]

• Protected Readonly Abstract **[toStringTag]**: string

Enforced symbol property containing the singular name of this model. The value of this property represents the repository which all instances of this model are considered to belong to. In Detail, the different operations committed through this model are derived from this singular name (and the corresponding pluralized form).

Example

Provide a valid symbol property:

```
import { Model } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

Defined in packages/data/src/model/model.ts:919

---

data.Model.changes

changes

• Protected Readonly **changes**: BehaviorSubject<M>

BehaviorSubject emitting every time this model instance experiences a mutation.

Defined in packages/data/src/model/model.ts:982

---

data.Model.entity

entity

• Protected get **entity**(): string

Accessor to the singular name of this model.

Returns string

Singular name of this model.

Defined in packages/data/src/model/model.ts:1001

---

data.Model.plural

plural

• Protected get **plural**(): string

Accessor to the pluralized name of this model.

Returns string

Pluralized name of this model.

Defined in packages/data/src/model/model.ts:1010

---

data.Model.static

static

• Protected Readonly **static**: Type<M>

Type-asserted alias for the static model context.

Defined in packages/data/src/model/model.ts:987

---

data.Model.type

type

• `Protected` `get` **type**(): `string`

Accessor to the raw name of this model.

Returns `string`

Raw name of this model.

Defined in packages/data/src/model/model.ts:1019

---

data.Model

Model

• **Model**: `Object`

Namespace containing types and interfaces to be used in conjunction with classes extending the abstract model base class. All the types and interfaces within this namespace are only applicable to classes extending the abstract model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in packages/data/src/model/model.ts:18

packages/data/src/model/model.ts:119

packages/data/src/model/model.ts:318

---

data.Model.Field

Field

T **Field**<`T`>: `string` & `Exclude`<keyof `T`, `Exclude`<keyof `Model`, `"id"` | `"created"` | `"modified"`»

Type alias for all fields, i.e., own enumerable properties, (excluding internally used ones) of classes extending the abstract model base class.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.

Defined in packages/data/src/model/model.ts:27

---

data.Model.Filter

Filter

T **Filter**<`T`>: `Params`<`T`>

Type alias referencing Params.

See

Params

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.

Defined in packages/data/src/model/model.ts:37

packages/data/src/model/model.ts:119

---

data.Model.Filter

Filter

• **Filter**: `Object`

Namespace containing types and interfaces to be used when searching through the repositories of classes extending the abstract model base class. All the interfaces within this namespace are only applicable to classes extending the abstract model base class, as their generic type argument is always constrained to this abstract base class.

See

Model

Defined in packages/data/src/model/model.ts:37

packages/data/src/model/model.ts:119

Model.Filter.Conjunction

Conjunction

T **Conjunction**: `"AND"` | `"AND_NOT"` | `"OR"` | `"OR_NOT"`

Type alias for a string union type of all possible filter conjunctions, namely: `'AND'`, `'AND_NOT'`, `'OR'` and `'OR_NOT'`.

Defined in packages/data/src/model/model.ts:125

Model.Filter.Expression

Expression

• **Expression**<`T`>: `Object`

Interface describing filter expressions which may be employed through the Params as part of a findAll invocation. Filter expressions can either be the plain shape of an `entity` or compositions of multiple filter expressions, conjunct by one of the Conjunctions.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.

Defined in packages/data/src/model/model.ts:153

Filter.Expression.conjunction

conjunction

• `Optional` `Readonly` **conjunction**: `Object`

Conjunction of multiple filter Expressions requested data models are matched against. The conjunction sibling parameter has to be undefined when supplying this parameter. By supplying filter expressions, conjunct by specific Conjunction operators, fine-grained filter operations can be compiled.

See

- Conjunction
- Expression

Type declaration

Name	Type	Description
`operands`	`Expression<T>[]`	List of Expressions which are logically combined through an operator. These expressions may be nested and can be used to construct complex composite filter operations. **See** - Expression - operator
`operator?`	`Conjunction`	Conjunction operator used to logically combine all supplied operands. **See** - Conjunction - operands

Defined in packages/data/src/model/model.ts:165

Filter.Expression.entity

entity

• `Optional` `Readonly` **entity**: `Object`

Shape the requested data models are matched against. Supplying this parameter requires the conjunction sibling parameter to be undefined. By specifying the shape to match data models against, simple filter operations can be compiled.

See

conjunction

Type declaration

Name	Type	Description
`operator?`	`Operator`	Filter Operator to use for matching. **See** Operator
`path`	`Path<T, []>`	Property path from within the data model which to match against. The value which will be matched against has to be supplied through the value property. **See** value
`value`	`unknown`	Property value to match data models against. The property path of this value has to be supplied through the path property. **See** path

Defined in packages/data/src/model/model.ts:196

Model.Filter.Operator

Operator

T **Operator**: `"EQUAL"` | `"GREATER_OR_EQUAL"` | `"GREATER_THAN"` | `"LESS_OR_EQUAL"` | `"LESS_THAN"` | `"LIKE"` | `"NOT_EQUAL"`

Type alias for a string union type of all possible filter operators, namely: `'EQUAL'`, `'NOT_EQUAL'`, `'LIKE'`, `'GREATER_THAN'`, `'GREATER_OR_EQUAL'`, `'LESS_THAN'` and `'LESS_OR_EQUAL'`.

Defined in

packages/data/src/model/model.ts:136

Model.Filter.Params

Params

• **Params**<T>: `Object`

Interface describing the parameters of, e.g., the findAll method. This is the most relevant interface within this namespace (and is therefore also referenced by the Filter type alias), as it describes the input parameters of any filter operation.

Type parameters

Name	Type	Description
T	extends Model	Extending model instance type.

Defined in

packages/data/src/model/model.ts:234

Filter.Params.dir

dir

• `Optional` `Readonly` **dir**: `"desc"` | `"asc"`

Desired sorting direction of the requested data models. To specify which field the results should be sorted by, the sort property must be supplied.

See

sort

Defined in

packages/data/src/model/model.ts:243

Filter.Params.expression

expression

• `Optional` `Readonly` **expression**: `Expression<T>`

Expression to evaluate results against. This expression may be a simple matching or more complex, conjunct and nested expressions.

See

Expression

Defined in

packages/data/src/model/model.ts:251

Filter.Params.page

page

• `Optional` `Readonly` **page**: `number`

Page number, i.e., offset within the list of all results for a data model request. This property should be used together with the page size property.

See

size

Defined in packages/data/src/model/model.ts:260

Filter.Params.search

search

• `Optional` `Readonly` **search**: `string`

Free-text search field. This field overrides all expressions, as such that if this field contains a value, all expressions are ignored and only this free-text search filter is applied.

See

expression

Defined in packages/data/src/model/model.ts:269

Filter.Params.size

size

• `Optional` `Readonly` **size**: `number`

Page size, i.e., number of results which should be included within the response to a data model request. This property should be used together with the page offset property.

See

page

Defined in packages/data/src/model/model.ts:278

Filter.Params.sort

sort

• `Optional` `Readonly` **sort**: `Path<T, []>`

Property path used to determine the value which to sort the requested data models by. This property should be used together with the sorting direction property.

See

dir

Defined in packages/data/src/model/model.ts:287

---

data.Model.Graph

Graph

T **Graph**<T>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? Record<K, Graph<I> | Function> : K }[`Field<T>`][]

Mapped type to compile strongly typed graphs of classes extending the abstract model base class, while providing intellisense.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.

Defined in packages/data/src/model/model.ts:45

---

data.Model.Path

Path

T **Path**<`T`, `S`>: { [K in Field<T>]: S extends Object ? never : Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> | Model<infer I>[] ? `${K}.{Path<I, [...S, string]>}` : K }[`Field<T>`]

Mapped type to compile strongly typed property paths of classes extending the abstract model base class, while providing intellisense.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.
`S`	extends `string`[] = []	String array limiting the path depth.

Defined in packages/data/src/model/model.ts:61

---

data.Model.Shape

Shape

T **Shape**<`T`>: { [K in Field<T>]?: Required<T>[K] extends Function ? never : Required<T>[K] extends Model<infer I> ? Shape<I> : Required<T>[K] extends Model<infer I>[] ? Shape<I>[] : Required<T>[K] }

Mapped type to compile strongly typed shapes of classes extending the abstract model base class, while providing intellisense.

Type parameters

Name	Type	Description
`T`	extends `Model`	Extending model instance type.

Defined in packages/data/src/model/model.ts:78

---

data.Model.Type

Type

• **Type**<`T`>: `Object`

Interface describing the type, i.e., static constructable context, of classes extending the abstract model base class.

Type parameters

Name	Type	Description
T	extends Model	Extending model instance type.

Defined in packages/data/src/model/model.ts:95

Model.Type.commit

commit

▸ **commit**<T>(this, operation, variables?): Observable<any>

Static commit method. Calling this method on a class extending the abstract model base class, while supplying an operation and all its embedded variables, will dispatch the supplied operation to the respective model repository through the highest priority Querier or, if no querier is compatible, throw an error. This method is the central point of origin for all model-related data transferral and is internally called by all other distinct methods of the model.

Throws

Observable of ReferenceError.

Example

Commit a query-type operation:

```
import { ExampleModel } from './example-model';

ExampleModel.commit(`query queryExample(variable: $variable) {
  result {
    field
  }
}`, {
  variable: 'value'
}).subscribe(console.log);
```

See

Querier

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
operation	‘mutation ${string}‘ | ‘query ${string}‘ | ‘subscription ${string}‘	Operation to commit.
variables	Variables	Variables within the operation.

Returns Observable<any>

Observable of the commitment.

Inherited from Required.commit

Defined in packages/data/src/model/model.ts:352

Model.Type.constructor

constructor

• **new Type**(...`args`)

Overridden and concretized constructor signature.

Parameters

Name	Type	Description
`...args`	`Shape<Model<any»[]`	Class constructor rest parameter.

Inherited from

Required<typeof Model>.constructor

Defined in

packages/data/src/model/model.ts:103

Model.Type.deleteAll

deleteAll

▸ **deleteAll**<`T`>(`this`, `uuids`): `Observable<any>`

Static deleteAll method. Calling this method on a class extending the model, while supplying a list of `uuids`, will dispatch the deletion of all model instances identified by these UUIDs to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-deletions from the respective model repository can be achieved.

`Example`

Delete all model instances by UUIDs:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteAll([
  'b050d63f-cede-46dd-8634-a80d0563ead8',
  'a0164132-cd9b-4859-927e-ba68bc20c0ae',
  'b3fca31e-95cd-453a-93ae-969d3b120712'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`uuids`	`string[]`	UUIDs of model instances to be deleted.

Returns

`Observable<any>`

Observable of the deletion.

Inherited from

Required.deleteAll

Defined in

packages/data/src/model/model.ts:403

Model.Type.deleteOne

deleteOne

▸ **deleteOne**<T>(this, uuid): Observable<any>

Static deleteOne method. Calling this method on a class extending the model, while supplying an uuid, will dispatch the deletion of the model instance identified by this UUID to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the deletion of a single model instance from the respective model repository can be achieved.

Example

Delete one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.deleteOne(
  '18f3aa99-afa5-40f4-90c2-71a2ecc25651'
).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
uuid	string	UUID of the model instance to be deleted.

Returns Observable<any>

Observable of the deletion.

Inherited from Required.deleteOne

Defined in packages/data/src/model/model.ts:437

Model.Type.findAll

findAll

▸ **findAll**<T>(this, filter, graph): Observable<{ result: T[] ; total: number }>

Static findAll method. Calling this method on a class extending the abstract model base class, while supplying a filter to match model instances by and a graph containing the fields to be included in the result, will dispatch the lookup operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the bulk-lookup of model instances from the respective model repository can be achieved.

Example

Find all UUIDs for model instances modified between two dates:

```
import { ExampleModel } from './example-model';

ExampleModel.findAll({
  expression: {
    conjunction: {
      operands: [
        {
          entity: {
            operator: 'GREATER_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-01-01')
```

```
          }
        },
        {
          entity: {
            operator: 'LESS_OR_EQUAL',
            path: 'modified',
            value: new Date('2021-12-12')
          }
        }
      ],
      operator: 'AND'
    }
  }
}, [
  'id',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
filter	Filter<T>	Filter to find model instances by.
graph	Graph<T>	Graph of fields to be included.

Returns `Observable<{ result: T[]; total: number }>`

Observable of the find operation.

Inherited from Required.findAll

Defined in packages/data/src/model/model.ts:496

Model.Type.findOne

findOne

▸ **findOne**<T>(this, shape, graph): Observable<T>

Static findOne method. Calling this method on a class extending the abstract model base class, while supplying the `shape` to match the model instance by and a `graph` describing the fields to be included in the result, will dispatch the lookup operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, the retrieval of one specific model instance from the respective model repository can be achieved.

Example

Find one model instance by UUID:

```
import { ExampleModel } from './example-model';

ExampleModel.findOne({
  id: '2cfe7609-c4d9-4e4f-9a8b-ad72737db48a'
}, [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
shape	Shape<T>	Shape of the model instance to find.
graph	Graph<T>	Graph of fields to be included.

Returns Observable<T>

Observable of the find operation.

Inherited from Required.findOne

Defined in packages/data/src/model/model.ts:544

Model.Type.prototype

prototype

• **prototype**: Model<any>

Inherited from Required.prototype

Model.Type.saveAll

saveAll

▸ **saveAll**<T>(this, models, graph): Observable<T[]>

Static saveAll method. Calling this method on a class extending the abstract model base class, while supplying a list of models which to save and a graph describing the fields to be included in the result, will dispatch the save operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, bulk-persistance of model instances from the respective model repository can be achieved.

Example

Persist multiple models:

```
import { ExampleModel } from './example-model';

ExampleModel.saveAll([
  new ExampleModel({ field: 'example_1' }),
  new ExampleModel({ field: 'example_2' }),
  new ExampleModel({ field: 'example_3' })
], [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
models	T[]	Array of models to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns Observable<T[]>

Observable of the save operation.

Inherited from Required.saveAll

Defined in packages/data/src/model/model.ts:590

Model.Type.saveOne

saveOne

▸ **saveOne**<T>(this, model, graph): Observable<T>

Static saveOne method. Calling this method on a class extending the abstract model base class, while supplying a model which to save and a graph describing the fields to be included in the result, will dispatch the save operation to the respective model repository by internally calling the commit operation with suitable arguments. Through this method, persistance of one specific model instance from the respective model repository can be achieved.

Example

Persist a model:

```
import { ExampleModel } from './example-model';

ExampleModel.saveOne(new ExampleModel({ field: 'example' }), [
  'id',
  'modified',
  'field'
]).subscribe(console.log);
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
this	Type<T>	Static polymorphic this.
model	T	Model which is to be saved.
graph	Graph<T>	Graph of fields to be included.

Returns Observable<T>

Observable of the save operation.

Inherited from Required.saveOne

Defined in packages/data/src/model/model.ts:632

Model.Type.serialize

serialize

▸ **serialize**<T>(this, model, shallow?): undefined | Shape<T>

Static serialize method. Calling this method on a class extending the model, while supplying a `model` which to serialize and optionally enabling `shallow` serialization, will return the shape of the model, i.e., a plain JSON representation of all model fields, or undefined, if the supplied `model` does not contain any fields or values. By serializing `shallowly`, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the serialization of one specific model instance from the respective model repository can be achieved.

Example

Serialize a model:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const shape = ExampleModel.serialize(model);
console.log(shape); // { field: 'example' }
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	Static polymorphic this.
model	T	undefined	Model which is to be serialized.
shallow	boolean	false	Whether to serialize shallowly.

Returns undefined | Shape<T>

Shape of the model or undefined.

Inherited from Required.serialize

Defined in packages/data/src/model/model.ts:674

Model.Type.treemap

treemap

▸ **treemap**<T>(this, model, shallow?): undefined | Graph<T>

Static treemap method. Calling this method on a class extending the abstract model base class, while supplying a `model` which to treemap and optionally enabling `shallow` treemapping, will return a graph describing the fields which are declared and defined on the supplied `model`, or undefined, if the supplied `model` does not contain any fields or values. By treemapping `shallowly`, only properties defined on the supplied `model` are included (which means, all one-to-one and one-to-many associations are ignored). Through this method, the graph for one specific model instance from the respective model repository can be retrieved.

Example

Treemap a model:

```ts
import { ExampleModel } from './example-model';

const model = new ExampleModel({ field: 'example' });
const graph = ExampleModel.treemap(model);
console.log(graph); // ['field']
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Default value	Description
this	Type<T>	undefined	Static polymorphic this.
model	T	undefined	Model which is to be treemapped.
shallow	boolean	false	Whether to treemap shallowly.

Returns undefined | Graph<T>

Graph of the model or undefined.

Inherited from Required.treemap

Defined in packages/data/src/model/model.ts:742

Model.Type.unravel

unravel

▸ **unravel**<T>(this, graph): string

Static unravel method. Calling this method on a class extending the abstract model base class, while supplying a graph describing the fields which to unravel, will return the unraveled graph as raw string. Through this method, the graph for one specific model instance from the respective model repository can be unraveled into a raw string. This unraveled graph can than be consumed by, e.g., the commit method.

Example

Unravel a graph:

```ts
import { ExampleModel } from './example-model';

const unraveled = ExampleModel.unravel([
  'id',
  'modified',
  'field'
]);

console.log(unraveled); // '{id modified field}'
```

Type parameters

Name	Type	Description
T	extends Model<any, T>	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`graph`	`Graph<T>`	Graph which is to be unraveled.

Returns `string`

Unraveled graph as raw string.

Inherited from Required.unravel

Defined in packages/data/src/model/model.ts:806

Model.Type.valuate

valuate

▸ **valuate**<T>(`this`, `model`, `field`): `any`

Static valuate method. Calling this method on a class extending the abstract model base class, while supplying a `model` and a `field` which to valuate, will return the preprocessed value (e.g., primitive representation of JavaScript Dates) of the supplied `field` of the supplied `model`. Through this method, the preprocessed `field` value of one specific model instance from the respective model repository can be retrieved.

`Example`

Valuate a field:

```
import { ExampleModel } from './example-model';

const model = new ExampleModel({ created: new Date(0) });
const value = ExampleModel.valuate(model, 'created');
console.log(value); // '1970-01-01T00:00:00.000+00:00'
```

Type parameters

Name	Type	Description
`T`	extends `Model<any, T>`	Extending model instance type.

Parameters

Name	Type	Description
`this`	`Type<T>`	Static polymorphic this.
`model`	`T`	Model which is to be valuated.
`field`	`Field<T>`	Field of the model to be valuated.

Returns `any`

Valuated field value.

Inherited from Required.valuate

Defined in packages/data/src/model/model.ts:875

---

data.Property

Property

T **Property**: `Type<any>` | typeof `Boolean` | typeof `Date` | typeof `Number` | typeof `String`

Type alias for a union type of all primitive constructors which may be used as `typeFactory` argument for the Property decorator.

See

Property

Defined in packages/data/src/relation/property.ts:61

packages/data/src/relation/property.ts:10

---

data.Property

Property

▸ **Property**<T>(`typeFactory`, `transient?`): <M>(`model`: `M`, `field`: `Field<M>`) => `void`

Model field decorator factory. Using this decorator, Models can be enriched with primitive fields. The compatible primitives are the subset of primitives JavaScript shares with JSON, i.e., `Boolean`, `Date` (in serialized form), `Number` and `String`. The `Object` primitive cannot be uses as a `typeFactory`, as Model fields containing objects are declared by the HasOne and HasMany Model field decorators. By employing this decorator, the decorated field will (depending on the `transient` argument) be recognized when serializing or treemapping the Model containing the decorated field.

Example

Model with a string type field:

```
import { Model, Property } from '@sgrud/data';

export class ExampleModel extends Model<ExampleModel> {

  @Property(() => String)
  public field?: string;

  protected [Symbol.toStringTag]: string = 'ExampleModel';

}
```

See

- Model
- HasOne
- HasMany

Type parameters

Name	Type	Description
`T`	extends `Property`	Field value constructor type.

Parameters

Name	Type	Default value	Description
`typeFactory`	() => `T`	`undefined`	Forward reference to the field value constructor.
`transient`	`boolean`	`false`	Whether the decorated field is transient.

Returns `fn`

Model field decorator.

▸ <M>(`model`, `field`): `void`

Type parameters

Name	Type
`M`	extends `Model<any, M>`

Parameters

Name	Type
`model`	`M`
`field`	`Field<M>`

Returns `void`

Defined in packages/data/src/relation/property.ts:61

---

data.Querier

Querier

• `Abstract` **Querier**: `Object`

Abstract base class to implement data queriers. By extending this abstract base class and providing the extending class to the Linker, e.g., by Targeting it, the respective classes priority method will be called whenever the commit method is invoked and, if this class claims the highest priority, its commit method will be called with an `operation` and all the `variables` embedded within this operation.

Example

Simple querier stub:

```
import type { Model, Querier } from '@sgrud/data';
import { Provider, Target } from '@sgrud/core';
import type { Observable } from 'rxjs';

@Target<typeof ExampleQuerier>()
export class ExampleQuerier
  extends Provider<typeof Querier>('sgrud.data.querier.Querier') {

  public override readonly types: Set<Querier.Type> = new Set<Querier.Type>([
    'query'
  ]);

  public override commit(
    operation: Querier.Operation,
    variables: Querier.Variables
  ): Observable<any> {
    throw new Error('Stub!');
  }

  public override priority(model: Model.Type<any>): number {
    return 0;
  }

}
```

Decorator

Provide

See

Model

Defined in packages/data/src/querier/querier.ts:13

packages/data/src/querier/querier.ts:83

---

data.Querier.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.data.querier.Querier"`

Magic string by which this class is provided.

See

provide

Defined in packages/data/src/querier/querier.ts:90

---

data.Querier.commit

commit

▸ `Abstract` **commit**(`operation`, `variables`): `Observable<any>`

The overridden commit method of Targeted queriers is called by the commit method to commit operations. The invocation arguments are the `operation`, unraveled into a string, and all `variables` embedded within this operation. The extending class has to serialize the Variables and transfer the operation. It's the callers responsibility to unravel the Operation prior to invoking this method, and to deserialize and error handle whatever response is received.

Parameters

Name	Type	Description
`operation`	'mutation ${string}' | 'query ${string}' | 'subscription ${string}'	Querier operation to be committed.
`variables`	`Variables`	Variables within the operation.

Returns `Observable<any>`

Observable of the committed operation.

Defined in packages/data/src/querier/querier.ts:113

---

data.Querier.constructor

constructor

• **new Querier**()

---

data.Querier.priority

priority

▸ `Abstract` **priority**(`model`): `number`

Whenever the commit method is invoked, all Targeted and compatible queriers, i.e., implementations of the Querier class capable of handling the specific Type of the to be committed Operation, will be asked to prioritize themselves regarding the respective `model`. The implementation claiming the highest priority will be chosen and called to commit the outstanding operation.

Parameters

Name	Type	Description
`model`	`Type<any>`	Model to be committed.

Returns `number`

Priority of this implementation.

Defined in packages/data/src/querier/querier.ts:129

---

data.Querier.types

types

• `Readonly` `Abstract` **types**: `Set<Type>`

A set containing the the Types this class can handle. May contain none to all of `'mutation'`, `'query'` and `'subscription'`.

Defined in packages/data/src/querier/querier.ts:97

---

data.Querier

Querier

• **Querier**: `Object`

Namespace containing types and interfaces to be used in conjunction with the abstract Querier base class and in context of the Model data handling.

See

- Model
- Querier

Defined in packages/data/src/querier/querier.ts:13

packages/data/src/querier/querier.ts:83

---

data.Querier.Operation

Operation

T **Operation**: ‘${Type} ${string}‘

String literal helper type. Enforces any assigned string to conform to the standard form of an operation: A string, starting with the Type, followed by one whitespace and the operation content.

Defined in packages/data/src/querier/querier.ts:29

---

data.Querier.Type

Type

T **Type**: `"mutation"` | `"query"` | `"subscription"`

Type alias for a string union type of all known Operation types: `'mutation'`, `'query'` and `'subscription'`.

Defined in packages/data/src/querier/querier.ts:19

---

data.Querier.Variables

Variables

• **Variables**: `Object`

Interface describing the shape of variables which may be embedded within Operations. Variables are a simple key-value map, which can be deeply nested.

Defined in packages/data/src/querier/querier.ts:36

---

data.enumerate

enumerate

▸ **enumerate**<T>(`enumerator`): `T`

Enumeration helper function. Enumerations are special objects and all used TypeScript enums have to be looped through this helper function before they can be utilized in conjunction with the Model.

Example

Enumerate an enum:

```
import { enumerate } from '@sgrud/data';

enum Enumeration {
  One = 'ONE',
  Two = 'TWO'
}

export type ExampleEnum = Enumeration;
export const ExampleEnum = enumerate(Enumeration);
```

See

Model

Type parameters

Name	Type	Description
`T`	extends `object`	TypeScript enumeration type.

Parameters

Name	Type	Description
`enumerator`	`T`	TypeScript enumeration.

Returns `T`

Processed enumeration.

Defined in packages/data/src/model/enum.ts:50

---

data.hasMany

hasMany

• `Const` **hasMany**: typeof `hasMany`

Symbol used as property key by the HasMany decorator to register decorated fields for further computation, e.g., serialization, treemapping etc.

See

HasMany

Defined in

packages/data/src/relation/has-many.ts:11

---

data.hasOne

hasOne

• `Const` **hasOne**: typeof `hasOne`

Symbol used as property key by the HasOne decorator to register decorated fields for further computation, e.g., serialization, treemapping etc.

See

HasOne

Defined in

packages/data/src/relation/has-one.ts:11

---

data.property

property

• `Const` **property**: typeof `property`

Symbol used as property key by the Property decorator to register decorated fields for further computation, e.g., serialization, treemapping etc.

See

Property

Defined in

packages/data/src/relation/property.ts:24

Module: shell

shell

• **shell**: `Object`

`@sgrud/shell` - The SGRUD web UI shell.

The functions and classes found within this module ease the implementation of component-based frontends by providing `jsx-runtime`-compliant bindings and a router targeted at rendering components based upon the `@sgrud` libraries, but not limited to those. Furthermore, complex routing strategies and actions may be implemented through interceptor-like router tasks.

Defined in

packages/shell/index.ts:13

---

shell.Attribute

Attribute

▸ **Attribute**(name?): (prototype: Component, propertyKey: PropertyKey) => void

Example

Decorate a property:

```
import { Attribute, Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Attribute()
  public field?: string;

  public get template(): JSX.Element {
    return <span>Attribute value: {this.field}</span>;
  }

}
```

See

Component

Parameters

Name	Type	Description
name?	string	Component attribute name.

Returns fn

Component property decorator.

▸ (prototype, propertyKey): void

Parameters

Name	Type
prototype	Component
propertyKey	PropertyKey

Returns void

Defined in packages/shell/src/component/attribute.ts:54

---

shell.Catch

Catch

T **Catch**: (error: any) => boolean | undefined

Type alias for the *CatchTask* representing a function that will be called with the thrown error. The boolean return value will be used to examine wether the component containing the decorated property is responsible to handle the thrown error.

See

CatchTask

Defined in packages/shell/src/task/catch.ts:35

packages/shell/src/task/catch.ts:17

---

shell.Catch

Catch

▸ **Catch**(`filter?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. Applying this decorator to a property, while optionally supplying a **filter**

`See`

CatchTask

Parameters

Name	Type
`filter?`	`Catch`

Returns `fn`

Prototype property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/task/catch.ts:35

---

shell.CatchTask

CatchTask

• **CatchTask**: `Object`

`Decorator`

Target

`Decorator`

Singleton

`See`

RouterTask

Defined in packages/shell/src/task/catch.ts:71

---

shell.CatchTask.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.router.RouterTask"`

Magic string by which this class is provided.

See

provide

Inherited from RouterTask.[provide]

Defined in packages/shell/src/router/task.ts:47

---

shell.CatchTask.constructor

constructor

• **new CatchTask**()

Overrides RouterTask.constructor

Defined in packages/shell/src/task/catch.ts:93

---

shell.CatchTask.handle

handle

▸ **handle**(`_prev`, `next`, `handler`): `Observable<State<string»`

Parameters

Name	Type	Description
`_prev`	`State<string>`	Previously active router state (ignored).
`next`	`State<string>`	Next router state to be activated.
`handler`	`Task`	Next task handler.

Returns `Observable<State<string»`

Next handled router state.

Overrides RouterTask.handle

Defined in packages/shell/src/task/catch.ts:106

---

shell.CatchTask.trapped

trapped

• `Readonly` **trapped**: `Map<Function, Record<PropertyKey, any»`

Defined in packages/shell/src/task/catch.ts:77

---

shell.CatchTask.traps

traps

• `Readonly` **traps**: `Map<Function, Map<PropertyKey, Catch»`

Defined in packages/shell/src/task/catch.ts:82

---

shell.CatchTask.handleErrors

handleErrors

▸ `Private` **handleErrors**(): `Observable<never>`

Returns `Observable<never>`

.

Defined in packages/shell/src/task/catch.ts:207

---

shell.CatchTask.lookup

lookup

▸ `Private` **lookup**(`selector`, `routes?`): `undefined` | `string`

Parameters

Name	Type	Description
`selector`	`string`	Component tag name.
`routes`	`Iterable<Route<string»`	Routes to use for resolving.

Returns `undefined` | `string`

Resolved route or `undefined`.

Defined in packages/shell/src/task/catch.ts:231

---

shell.CatchTask.router

router

• `Private` `Readonly` **router**: `Router`

`Decorator`

Factor

Defined in packages/shell/src/task/catch.ts:88

---

shell.Component

Component

▸ **Component**<`S`, `R`>(`selector`, `inherits?`): <T>(`constructor`: `T`) => `T`

Class decorator factory. Registers the decorated class as component through the customElements registry. Registered components can be used in conjunction with the Attribute and Reference prototype property decorators which will trigger the component to re-render, when one of the observedAttributes or observedReferences changes. While any custom component which is registered by this decorator is enriched with basic rendering functionality, any implemented method will cancel out its `super` logic.

`Example`

Register a component:

```
import { Component } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  public readonly styles: string[] = [`
    span {
      font-style: italic;
    }
  `];

  public get template(): JSX.Element {
    return <span>Example component</span>;
  }

}
```

See

- Attribute
- Reference

Type parameters

Name	Type	Description
`S`	extends `CustomElementTagName`	Component tag type.
`R`	extends `HTMLElementTagName`	-

Parameters

Name	Type	Description
`selector`	`S`	Component tag name.
`inherits?`	`R`	Extended tag name.

Returns `fn`

Class decorator.

▸ <T>(`constructor`): `T`

Type parameters

Name	Type
`T`	extends () => `Component` & `HTMLElementTagNameMap[S]` & `HTMLElementTagNameMap[R]`

Parameters

Name	Type
`constructor`	`T`

Returns `T`

Defined in packages/shell/src/component/component.ts:138

---

shell.Component

Component

• **Component**: `Object`

Interface describing the shape of a custom component. Mostly adheres to the specs while providing rendering and change detection capabilities.

Defined in packages/shell/src/component/component.ts:138

packages/shell/src/component/component.ts:9

---

shell.Component.adoptedCallback

adoptedCallback

▸ `Optional` **adoptedCallback**(): `void`

Called when the component is moved between documents.

Returns `void`

Defined in packages/shell/src/component/component.ts:48

---

shell.Component.attributeChangedCallback

attributeChangedCallback

▸ `Optional` **attributeChangedCallback**(`name`, `prev?`, `next?`): `void`

Called when one of the component's observed attributes is added, removed or changed. Which component attributes are observed depends on the contents of the observedAttributes array.

Parameters

Name	Type	Description
`name`	`string`	Attribute name.
`prev?`	`string`	Previous value.
`next?`	`string`	Next value.

Returns `void`

Defined in packages/shell/src/component/component.ts:59

---

shell.Component.connectedCallback

connectedCallback

▸ `Optional` **connectedCallback**(): `void`

Called when the component is appended to or moved within the dom.

Returns `void`

Defined in packages/shell/src/component/component.ts:64

---

shell.Component.constructor

constructor

• **constructor**: `Object`

Inherited from HTMLElement.constructor

---

shell.Component.disconnectedCallback

disconnectedCallback

▸ `Optional` **disconnectedCallback**(): `void`

Called when the component is removed from the dom.

Returns `void`

Defined in packages/shell/src/component/component.ts:69

---

shell.Component.observedAttributes

observedAttributes

• `Optional` `Readonly` **observedAttributes**: `string[]`

Array of attribute names, which should be observed for changes, which will trigger the attributeChanged-Callback.

See

Attribute

Defined in packages/shell/src/component/component.ts:17

---

shell.Component.observedReferences

observedReferences

• `Optional` `Readonly` **observedReferences**: `Record<Key, keyof HTMLElementEventMap[]>`

Mapping of references to observed events, which, when emitted by the referenced node, trigger the referenceChangedCallback.

See

Reference

Defined in packages/shell/src/component/component.ts:25

---

shell.Component.readyState

readyState

• `Optional` `Readonly` **readyState**: `boolean`

Internal readiness indication. Initially resolves to `undefined` and will mirror the `isConnected` state, when ready.

Defined in packages/shell/src/component/component.ts:31

---

shell.Component.referenceChangedCallback

referenceChangedCallback

▸ `Optional` **referenceChangedCallback**(`name`, `node`, `event`): `void`

Called when one of the component's referenced and observed nodes emits an event. Which referenced nodes are observed for which events depends on the contents of the observedReferences mapping.

Parameters

Name	Type	Description
`name`	`string`	Reference name.
`node`	`Node`	-
`event`	`Event`	Emitted event.

Returns `void`

Defined in packages/shell/src/component/component.ts:79

---

shell.Component.renderComponent

renderComponent

▸ `Optional` **renderComponent**(): `void`

Called when the component has changed and should be (re-)rendered.

Returns `void`

Defined in packages/shell/src/component/component.ts:84

---

shell.Component.styles

styles

• `Optional` `Readonly` **styles**: `string`[]

Array of CSS strings, which should be included within the shadow dom of the component.

Defined in packages/shell/src/component/component.ts:37

---

shell.Component.template

template

• `Optional` `Readonly` **template**: `Element`

JSX representation of the component template. If no template is supplied, a slot element will be rendered instead.

Defined in packages/shell/src/component/component.ts:43

---

shell.CustomElementTagName

CustomElementTagName

T **CustomElementTagName**: `Extract`<keyof `HTMLElementTagNameMap`, ‘$string—${string}‘>

Global string literal helper type. Enforces any assigned string to be a `keyof HTMLElementTagNameMap`, while excluding built-in tag names, i.e., extracting all `${string}-${string}` keys of `HTMLElementTagNameMap`.

`Example`

A valid `CustomElementTagName`:

```
const tagName: CustomElementTagName = 'example-component';
```

Defined in

packages/shell/src/component/runtime.ts:17

---

shell.HTMLElementTagName

HTMLElementTagName

T **HTMLElementTagName**: `Exclude`<keyof `HTMLElementTagNameMap`, ‘$string—${string}‘>

Global string literal helper type. Enforces any assigned string to be a `keyof HTMLElementTagNameMap`, while excluding custom element tag names, i.e., all `${string}-${string}` keys of `HTMLElementTagNameMap`.

`Example`

A valid `HTMLElementTagNameMap`:

```
const tagName: HTMLElementTagNameMap = 'div';
```

Defined in

packages/shell/src/component/runtime.ts:31

---

shell.JSX

JSX

• **JSX**: `Object`

Intrinsic JSX namespace.

`See`

JSX

Defined in

packages/shell/src/component/runtime.ts:39

---

shell.JSX.Element

Element

T **Element**: () => `Node`[]

Intrinsic JSX element type helper representing an array of bound **incremental-dom** calls.

`See`

incremental-dom

Defined in

packages/shell/src/component/runtime.ts:47

---

shell.JSX.IntrinsicElements

IntrinsicElements

T **IntrinsicElements**: { [K in keyof HTMLElementTagNameMap]: Partial<HTMLElementTagNameMap[K]> & Object }

Intrinsic list of known JSX elements, comprised of the global `HTMLElementTagNameMap`.

Defined in

packages/shell/src/component/runtime.ts:53

---

shell.JSX.Key

Key

T **Key**: `string` | `number`

Element reference key type helper. Enforces any assigned value to adhere to the **incremental-dom** `Key` type.

See

incremental-dom

Defined in

packages/shell/src/component/runtime.ts:77

---

shell.Reference

Reference

▸ **Reference**(`ref`, `observe?`): (`prototype`: `Component`, `propertyKey`: `PropertyKey`) => `void`

Prototype property decorator factory. Applying this decorator to a property of a registered Component while supplying the `reference` key and, optionally, an array of events to `observe`, will replace the decorated property with a getter returning the referenced node, once rendered. If an array of events is supplied, whenever one of those events is emitted by the referenced node, the referenceChangedCallback is called with the `reference` key, the referenced node and the emitted event.

Example

Reference a node:

```
import { Component, Reference } from '@sgrud/shell';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Reference('example-key')
  private readonly span?: HTMLSpanElement;

  public get template(): JSX.Element {
    return <span key="example-key"></span>;
  }

}
```

See

Component

Parameters

Name	Type	Description
`ref`	`Key`	Element reference.

Name	Type	Description
`observe?`	keyof `HTMLElementEventMap[]`	Events to observe.

Returns `fn`

Component property decorator.

▸ (`prototype`, `propertyKey`): `void`

Parameters

Name	Type
`prototype`	`Component`
`propertyKey`	`PropertyKey`

Returns `void`

Defined in packages/shell/src/component/reference.ts:46

---

shell.Resolve

Resolve

T **Resolve**<`S`>: (`segment`: `Segment<S>`, `state`: `State`) => `Observable<any>`

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	Route path string type.

Type declaration ▸ (`segment`, `state`): `Observable<any>`

Type alias for the ResolveTask representing a function that will be called with the respective Segment and State.

See

ResolveTask

Parameters

Name	Type
`segment`	`Segment<S>`
`state`	`State`

Returns `Observable<any>`

Defined in packages/shell/src/task/resolve.ts:67

packages/shell/src/task/resolve.ts:16

---

shell.Resolve

Resolve

▸ **Resolve**<S>(task): (prototype: Component, propertyKey: PropertyKey) => void

Prototype property decorator factory. Applying this decorator to a property of a registered Component while supplying a task to resolve will replace the decorated property with a getter returning the value the supplied task resolves to.

Example

Decorate a property:

```
import { Component, Resolve } from '@sgrud/shell';
import { of } from 'rxjs';

declare global {
  interface HTMLElementTagNameMap {
    'example-component': ExampleComponent;
  }
}

@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component {

  @Resolve((segment, state) => of([segment.route.path, state.search]))
  public readonly resolved!: [string, string];

  public get template(): JSX.Element {
    return <span>Resolved: {this.resolved.join('?')}</span>;
  }

}
```

See

ResolveTask

Type parameters

Name	Type	Description
S	extends string	Route path string type.

Parameters

Name	Type	Description
task	Resolve<S>	Task to resolve.

Returns fn

Prototype property decorator.

▸ (prototype, propertyKey): void

Parameters

Name	Type
prototype	Component
propertyKey	PropertyKey

Returns void

Defined in packages/shell/src/task/resolve.ts:67

---

shell.ResolveTask

ResolveTask

• **ResolveTask**: `Object`

Built-in RouterTask intercepting all navigational events of the Router to resolve Resolve tasks before invoking subsequent RouterTasks.

`Decorator`

Target

`Decorator`

Singleton

`See`

RouterTask

Defined in packages/shell/src/task/resolve.ts:107

---

shell.ResolveTask.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.router.RouterTask"`

Magic string by which this class is provided.

`See`

provide

Inherited from RouterTask.[provide]

Defined in packages/shell/src/router/task.ts:47

---

shell.ResolveTask.constructor

constructor

• **new ResolveTask**()

Public constructor. Called by the Target decorator to link this RouterTask into the Router.

Overrides RouterTask.constructor

Defined in packages/shell/src/task/resolve.ts:126

---

shell.ResolveTask.handle

handle

▸ **handle**(`_prev`, `next`, `handler`): `Observable<State<string»`

Overridden handle method of the RouterTask base class. Iterates all segments of the `next` State and collects all Resolve tasks for encountered Components in those segments. The collected tasks are resolved before invoking the subsequent RouterTask.

Name	Type	Description

Parameters

Name	Type	Description
`_prev`	`State<string>`	Previously active router state (ignored).
`next`	`State<string>`	Next router state to be activated.
`handler`	`Task`	Next task handler.

Returns `Observable<State<string»`

Next handled router state.

Overrides RouterTask.handle

Defined in packages/shell/src/task/resolve.ts:145

---

shell.ResolveTask.required

required

• `Readonly` **required**: `Map<Function, Map<PropertyKey, Resolve<string»>`

Mapping of all Components to a map of property keys and their corresponding Resolve tasks.

Defined in packages/shell/src/task/resolve.ts:114

---

shell.ResolveTask.resolved

resolved

• `Readonly` **resolved**: `Map<Function, Record<PropertyKey, any»`

Mapping of all Components to an object consisting of property keys and their corresponding Resolve tasks return values.

Defined in packages/shell/src/task/resolve.ts:120

---

shell.Route

Route

▸ **Route**<S>(`config`): <T>(`constructor`: `T`) => `void`

Class decorator factory. Applying this decorator to a custom component will associate the supplied route `configuration` to the decorated component constructor. Further, the `configured` children are iterated and every child that is a custom element itself will be replaced by its respective route `configuration`. Finally, the processed `configuration` for the decorated component is associated to the component constructor and added to the Router.

`Example`

Associate a route `configuration` to a component:

```
import { Component, Route } from '@sgrud/shell';
import { ChildComponent } from './child-component';

@Route({
  path: 'example',
  children: [
    ChildComponent
```

```
  ]
})
@Component('example-component')
export class ExampleComponent extends HTMLElement implements Component { }
```

See

Router

Type parameters

Name	Type	Description
`S`	extends `string`	Route path string type.

Parameters

Name	Type	Description
`config`	`Assign<{ children?: (Route<string> | CustomElementConstructor & { [route]?: Route<string> })[]; slots?: Record<string, CustomElementConstructor | CustomElementTagName> }, Omit<Route<S>, "component"» & { parent?: Route<string> | CustomElementConstructor & { [route]?: Route<string> } }`	Route configuration for this component.

Returns `fn`

Class decorator.

▸ <T>(`constructor`): `void`

Type parameters

Name	Type
`T`	extends `CustomElementConstructor & { [route]?: Route<S> }`

Parameters

Name	Type
`constructor`	`T`

Returns `void`

Defined in packages/shell/src/router/route.ts:95

---

shell.Route

Route

• **Route**<S>: `Object`

Interface describing the shape of a route. A route must consist of at least a path and may declare a component, which will be rendered when the route is navigated to, as well as slots and components which will be slotted within those. Furthermore a route may specify children.

`Example`

Define a route:

```ts
import type { Route } from '@sgrud/shell';

const route: Route = {
  path: '',
  component: 'example-component',
  children: [
    {
      path: 'child',
      component: 'child-component'
    }
  ]
};
```

`See`

Router

Type parameters

Name	Type	Description
S	extends `string` = `string`	Route path string type.

Defined in

packages/shell/src/router/route.ts:95

packages/shell/src/router/route.ts:32

---

shell.Route.children

children

• `Optional` `Readonly` **children**: `Route`<`string`>[]

Optional array of children for this route.

Defined in

packages/shell/src/router/route.ts:37

---

shell.Route.component

component

• `Optional` `Readonly` **component**: `CustomElementTagName`

Optional route component.

Defined in

packages/shell/src/router/route.ts:42

---

shell.Route.constructor

constructor

• **constructor**: `Object`

---

shell.Route.path

path

• `Readonly` **path**: `S`

Required route path.

Defined in packages/shell/src/router/route.ts:47

---

shell.Route.slots

slots

• `Optional` `Readonly` **slots**: `Record<string, CustomElementTagName>`

Optional mapping of slot names to slotted components.

Defined in packages/shell/src/router/route.ts:52

---

shell.Router

Router

• **Router**: `Object`

Targeted singleton Router class extending `Set<Route>`. This singleton class provides routing and rendering capabilities. Routing is primarily realized by maintaining (inheriting) a set of routes and (recursively) matching paths against those routes, when instructed so by calling navigate. When a matching Segment is found, the corresponding components are rendered by the handle method (part of the Task contract).

`Decorator`

Target

`Decorator`

Singleton

Defined in packages/shell/src/router/router.ts:13

packages/shell/src/router/router.ts:144

---

shell.Router.[observable]

[observable]

• `Readonly` **[observable]**: () => `Subscribable<State<string»`

Type declaration ▸ (): `Subscribable<State<string»`

Symbol property typed as callback to a Subscribable. The returned Subscribable emits the current State and every time this changes.

`Example`

Subscribe to the router:

```ts
import { Linker } from '@sgrud/core';
import { Router } from '@sgrud/shell';
import { from } from 'rxjs';

const router = new Linker<typeof Router>().get(Router);
from(router).subscribe(console.log);
```

Returns `Subscribable<State<string»`

Callback to a Subscribable.

Defined in packages/shell/src/router/router.ts:164

---

shell.Router.add

add

▸ **add**(`route`): `Router`

Overridden `Set.prototype.add` method. Invoking this method while supplying a `route` will add the supplied `route` to the router after deleting its child routes from the router, thereby ensuring that only topmost/root `routes` remain part of the router.

Parameters

Name	Type	Description
`route`	`Route<string>`	Route to add to the set.

Returns `Router`

This router instance.

Overrides Set.add

Defined in packages/shell/src/router/router.ts:234

---

shell.Router.baseHref

baseHref

• `Readonly` **baseHref**: `string`

Absolute base href for navigation.

Defined in packages/shell/src/router/router.ts:169

---

shell.Router.bind

bind

▸ **bind**(`this`, `outlet?`, `baseHref?`, `hashBased?`): `void`

Binding helper method. Calling this method will bind a handler to the `window.onpopstate` event, invoking navigate with the state of the `PopStateEvent`. This method furthermore allows the default and readonly properties baseHref, hashBased and outlet to be overridden. This method throws an error if called more than once, without calling the unbind method in between.

Parameters

Name	Type	Description
this	Mutable<Router>	Mutable polymorphic this.
outlet	Element | DocumentFragment	Navigated route rendering outlet.
baseHref	string	Absolute base href for navigation.
hashBased	boolean	Wether to employ hash-based routing.

Returns `void`

Defined in packages/shell/src/router/router.ts:257

---

shell.Router.constructor

constructor

• **new Router**()

Singleton router class constructor. This constructor is called once by the Target decorator and sets initial values on the instance. All subsequent calls will return the previously constructed singleton instance.

Overrides Set<Route>.constructor

Defined in packages/shell/src/router/router.ts:206

---

shell.Router.handle

handle

▸ **handle**(state, replace?): Observable<State<string»

Implementation of the handle method. This method is called internally by the match method after all RouterTasks have been invoked. It is therefore considered the default or fallback RouterTask and handles the rendering of the supplied state.

Parameters

Name	Type	Default value	Description
state	State<string>	undefined	Router state to handle.
replace	boolean	false	Wether to replace the state.

Returns Observable<State<string»

Observable of the handled state.

Implementation of Task.handle

Defined in packages/shell/src/router/router.ts:292

---

shell.Router.hashBased

hashBased

• Readonly **hashBased**: boolean

Wether to employ hash-based routing.

Defined in packages/shell/src/router/router.ts:174

---

shell.Router.join

join

▸ **join**(segment): `string`

Segment joining helper. The supplied `segment` is converted to a path by spooling to its top-most parent and iterating through all children while concatenating every encountered path. If said path represents a (optional) parameter, this portion of the path is replaced by the respective Segment.params value.

Parameters

Name	Type	Description
`segment`	`Segment<string>`	Segment to be joined.

Returns `string`

Joined segment as string.

Defined in packages/shell/src/router/router.ts:337

---

shell.Router.match

match

▸ **match**(path, routes?): `undefined` | `Segment<string>`

Main router matching method. Calling this method while supplying a `path` and optionally an array of `routes` will return a matching Segment or undefined, if no match was found. If no `routes` are supplied, routes previously added to the router set will be used. This method represents the backbone of the router, as it, given a list of `routes` and a `path`, will determine wether this path hits a match within the list of `routes`, thereby effectively determining navigational integrity.

`Example`

Test if path `'example/route'` matches `child` or `route`:

```
import { Router } from '@sgrud/shell';

const path = 'example/route';
const router = new Router();

const child = {
  path: 'route'
};

const route = {
  path: 'example',
  children: [child]
};

if (router.match(path, [child])) {
  // false
}

if (router.match(path, [route])) {
  // true
}
```

Parameters

Name	Type	Description
`path`	`string`	Path to match against.
`routes`	`Route<string>[]`	Routes to use for matching.

Returns `undefined | Segment<string>`

Matching segment or undefined.

Defined in packages/shell/src/router/router.ts:400

---

shell.Router.navigate

navigate

▸ **navigate**(`target`, `search?`, `replace?`): `Observable<State<string»`

Main navigation method. Calling this method while supplying either a path or Segment as navigation `target` (and optional `search` parameters) will normalize the path by trying to match a respective Segment or directly use the supplied Segment as next State. This upcoming state is looped through all linked RouterTasks and finally handled by the router itself to render the resulting, possibly intercepted and mutated state.

Parameters

Name	Type	Default value	Description
`target`	`string	Segment<string>`	`undefined`	Path or segment to navigate to.
`search?`	`string`	`undefined`	Optional search parameters.
`replace`	`boolean`	`false`	Wether to replace the state.

Returns `Observable<State<string»`

Observable of the router state.

Defined in packages/shell/src/router/router.ts:493

---

shell.Router.outlet

outlet

• `Readonly` **outlet**: `Element | DocumentFragment`

Navigated route rendering outlet.

Defined in packages/shell/src/router/router.ts:179

---

shell.Router.rebase

rebase

▸ **rebase**(`path`, `prefix?`): `string`

Rebasing helper method. Rebases the supplied `path` against the router baseHref, by either prepending the baseHref to the supplied `path` or stripping it, depending on the `prefix` argument.

Parameters

Name	Type	Default value	Description
`path`	`string`	`undefined`	Path to rebase against the baseHref.
`prefix`	`boolean`	`true`	Wether to prepend or strip the baseHref.

Returns `string`

Rebased `path`.

Defined in packages/shell/src/router/router.ts:550

---

shell.Router.spool

spool

▸ **spool**(`segment`, `rewind?`): `Segment<string>`

Spooling helper method. Given a `segment` (and wether to `rewind`), the top-most parent (or deepest child) of the graph-link Segment is returned.

Parameters

Name	Type	Default value	Description
`segment`	`Segment<string>`	`undefined`	Segment to spool.
`rewind`	`boolean`	`true`	Spool direction.

Returns `Segment<string>`

Spooled segment.

Defined in packages/shell/src/router/router.ts:575

---

shell.Router.state

state

• `get` **state**(): `State<string>`

Getter mirroring the current value of the changes behavior subject.

Returns `State<string>`

Defined in packages/shell/src/router/router.ts:197

---

shell.Router.unbind

unbind

▸ **unbind**(`this`): `void`

Unbinding helper method. Calling this method (after calling bind) will unbind the previously bound handler from the `window.onpopstate` event. Further, the arguments passed to bind are revoked, meaning the default values of the properties baseHref, hashBased and outlet are restored. Calling this method without previously binding the router will throw an error.

Parameters

Name	Type	Description
`this`	`Mutable<Router>`	Mutable polymorphic this.

Returns `void`

Defined in packages/shell/src/router/router.ts:602

---

shell.Router.changes

changes

• `Private` `Readonly` **changes**: `BehaviorSubject<State<string»`

Internally used behavior subject containing and emitting every navigated State.

Defined in packages/shell/src/router/router.ts:185

---

shell.Router

Router

• **Router**: `Object`

Namespace containing types and interfaces to be used in conjunction with the singleton Router class.

See

Router

Defined in packages/shell/src/router/router.ts:13

packages/shell/src/router/router.ts:144

---

shell.Router.Left

Left

T **Left**<`S`>: `S` extends ‘$infer L$/{string}‘ ? `L` : `S`

String literal helper type. Represents the leftest part of a path.

Type parameters

Name	Type	Description
`S`	extends `string`	Route path string type.

Defined in packages/shell/src/router/router.ts:20

---

shell.Router.Params

Params

T **Params**<S>: S extends ‘$string$:{infer P}‘ ? P extends ‘$Left < P$ {infer R}‘ ? Params<R> : never & Left<P> extends ‘${infer O}?‘ ? { [K in O]?: string } : { [K in Left<P>]: string } : {}

Type helper representing the (optional) parameters of a path. By extracting string literals starting with a colon (and optionally ending on a question mark), a union type of a key/value pair for each parameter is created.

Example

Extract parameters from path 'item/:id/field/:name?':

```ts
import type { Router } from '@sgrud/shell';

let params: Router.Params<'item/:id/field/:name?'>;
// { id: string; name?: string; }
```

Type parameters

Name	Description
S	Route path string type.

Defined in packages/shell/src/router/router.ts:38

---

shell.Router.Segment

Segment

• **Segment**<S>: Object

Interface describing the shape of a router segment. A segment represents a navigated Route and its corresponding Params. As routes are represented in a tree-like structure and one segment represents one layer within the route-tree, each segment may have a parent and/or a child. The resulting graph of segments represents the navigated path through the underlying route-tree.

Type parameters

Name	Type	Description
S	extends string = string	Route path string type.

Defined in packages/shell/src/router/router.ts:58

Router.Segment.child

child

• Optional Readonly **child**: Segment<string>

Optional child of this segment.

Defined in packages/shell/src/router/router.ts:63

Router.Segment.params

params

• Readonly **params**: Params<S>

Route path parameters and corresponding values.

Defined in packages/shell/src/router/router.ts:68

Router.Segment.parent

parent

• `Optional` `Readonly` **parent**: `Segment<string>`

Optional parent of this segment.

Defined in packages/shell/src/router/router.ts:73

Router.Segment.route

route

• `Readonly` **route**: `Route<S>`

Route associated to this segment.

Defined in packages/shell/src/router/router.ts:78

---

shell.Router.State

State

• **State**<`S`>: `Object`

Interface describing the shape of a router state. Router states correspond to history states, as each navigation results in a new state being created. Each navigated state is represented by the absolute navigated path, a Segment as entrypoint to the graph-like representation of the navigated path through the route-tree and search parameters.

Type parameters

Name	Type	Description
`S`	extends `string` = `string`	Route path string type.

Defined in packages/shell/src/router/router.ts:91

Router.State.path

path

• `Readonly` **path**: `S`

Absolute path of the router state.

Defined in packages/shell/src/router/router.ts:96

Router.State.search

search

• `Readonly` **search**: `string`

Search parameters of the router state.

Defined in packages/shell/src/router/router.ts:101

Router.State.segment

segment

• `Readonly` **segment**: `Segment<S>`

Segment of the router state.

Defined in packages/shell/src/router/router.ts:106

---

shell.Router.Task

Task

• **Task**: `Object`

Interface describing the shape of a RouterTask. These tasks are run whenever a navigation is triggered and may intercept and mutate the next state or completely block or redirect a navigation.

See

RouterTask

Defined in packages/shell/src/router/router.ts:117

Router.Task.handle

handle

▸ **handle**(`next`): `Observable<State<string»`

Method called when a navigation was triggered.

Parameters

Name	Type	Description
`next`	`State<string>`	Next state to be handled.

Returns `Observable<State<string»`

Observable of handled state.

Defined in packages/shell/src/router/router.ts:125

---

shell.RouterLink

RouterLink

• **RouterLink**: `Object`

Custom component extending the `HTMLAnchorElement`. This component provides a declarative way to invoke the Router, while maintaining compatibility with SSR/SEO aspects of SPAs. This is achieved by rewriting absolute `hrefs` to be contained within the baseHref and replacing the default browser behavior when onclicked with navigate.

Example

A `router-link`:

```
<a href="/example" is="router-link">Example</a>
```

See

- Route
- Router

Defined in packages/shell/src/router/link.ts:32

---

shell.RouterLink.observedAttributes

observedAttributes

■ `Static` `Readonly` **observedAttributes**: `string`[]

Array of attribute names, which should be observed for changes, which will trigger the attributeChanged-Callback. This component only observes the `href` attribute.

Defined in packages/shell/src/router/link.ts:39

---

shell.RouterLink.attributeChangedCallback

attributeChangedCallback

▸ **attributeChangedCallback**(`_name`, `_prev?`, `next?`): `void`

Called when the component's `href` attribute is added, removed or changed. The `next` attribute value is used to determine wether to rewrite the `href` by passing it through the rebase method.

Parameters

Name	Type	Description
`_name`	`string`	Attribute name (ignored).
`_prev?`	`string`	Previous value (ignored).
`next?`	`string`	Next value.

Returns `void`

Defined in packages/shell/src/router/link.ts:70

---

shell.RouterLink.constructor

constructor

• **new RouterLink**()

Overrides HTMLAnchorElement.constructor

Defined in packages/shell/src/router/link.ts:51

---

shell.RouterLink.onclick

onclick

• **onclick**: (`event`: `MouseEvent`) => `void`

Type declaration ▸ (`event`): `void`

Overridden `onclick` handler, preventing the default browser behavior and calling the navigate method instead.

Parameters

Name	Type	Description
`event`	`MouseEvent`	Mouse click event.

Returns `void`

Overrides HTMLAnchorElement.onclick

Defined in packages/shell/src/router/link.ts:87

---

shell.RouterLink.router

router

• `Private` `Readonly` **router**: `Router`

Factored-in router property retrieving the linked Router.

`Decorator`

Factor

Defined in packages/shell/src/router/link.ts:49

---

shell.RouterOutlet

RouterOutlet

• **RouterOutlet**: `Object`

Custom element extending the `HTMLSlotElement`. When this component is constructed, it binds the Router to itself while supplying the value of its baseHref attribute as baseHref and the presence of a hashBased attribute on itself as hashBased. This component should only be used once, as it will be used by the Router as outlet to render the current State.

`Example`

A `router-outlet`:

```
<slot baseHref="/example" is="router-outlet">Loading...</slot>
```

`See`

- Route
- Router

Defined in packages/shell/src/router/outlet.ts:33

---

shell.RouterOutlet.baseHref

baseHref

• `get` **baseHref**(): `undefined` | `string`

Getter mirroring the `baseHref` attribute of the component.

Returns `undefined` | `string`

Defined in packages/shell/src/router/outlet.ts:38

---

shell.RouterOutlet.constructor

constructor

• **new RouterOutlet**()

Custom `router-outlet` component constructor. Invokes bind on itself while supplying its baseHref attribute value and the presence of a hashBased attribute on itself. It furthermore invokes a `setTimeout` loop running until the number of routes the router contains evaluates truthy, which in turn triggers an initial navigate invocation.

Overrides HTMLSlotElement.constructor

Defined in packages/shell/src/router/outlet.ts:57

---

shell.RouterOutlet.hashBased

hashBased

• `get` **hashBased**(): `boolean`

Getter mirroring the presence of a `hashBased` attribute on the component.

Returns `boolean`

Defined in packages/shell/src/router/outlet.ts:45

---

shell.RouterTask

RouterTask

• `Abstract` **RouterTask**: `Object`

Abstract base class to implement Tasks. By Targeting or otherwise providing an implementation of this abstract base class to the Linker, the implemented handle method is called whenever a new State is triggered by navigating. This interceptor-pattern makes complex routing strategies like asynchronous module-retrieval and the like easy to be implemented.

Example

Simple router task stub:

```
import { Provider, Target } from '@sgrud/core';
import type { Router, RouterTask } from '@sgrud/shell';
import type { Observable } from 'rxjs';

@Target<typeof ExampleRouterTask>()
export class ExampleRouterTask
  extends Provider<typeof RouterTask>('sgrud.shell.router.RouterTask') {

  public override handle(
    prev: Router.State,
    next: Router.State,
    handler: Router.Task
  ): Observable<Router.State> {
    throw new Error('Stub!');
  }

}
```

Decorator

Provide

See

- Route
- Router

Defined in packages/shell/src/router/task.ts:40

---

shell.RouterTask.[provide]

[provide]

■ `Static` `Readonly` **[provide]**: `"sgrud.shell.router.RouterTask"`

Magic string by which this class is provided.

See

provide

Defined in packages/shell/src/router/task.ts:47

---

shell.RouterTask.constructor

constructor

• **new RouterTask**()

---

shell.RouterTask.handle

handle

▸ `Abstract` **handle**(`prev`, `next`, `handler`): `Observable<State<string>>`

Abstract handle method, called whenever a new State should be navigated to. This method provides the possibility to intercept these upcoming states and, e.g., mutate or redirect them.

Parameters

Name	Type	Description
`prev`	`State<string>`	Previously active router state.
`next`	`State<string>`	Next router state to be activated.
`handler`	`Task`	Next task handler.

Returns `Observable<State<string>>`

Next handled router state.

Defined in packages/shell/src/router/task.ts:60

---

shell.component

component

• `Const` **component**: typeof `component`

Defined in packages/shell/src/component/component.ts:91

---

shell.createElement

createElement

▸ **createElement**(`type`, `props?`, `ref?`): `Element`

JSX element factory. Provides `jsx-runtime`-compliant bindings to the incremental-dom library. This factory function is meant to be implicitly imported by the transpiler and returns an array of bound `incremental-dom` function calls, representing the created JSX element. This array of bound functions can be rendered into an element attached to the DOM through the `render` function.

See

- render
- incremental-dom

Parameters

Name	Type	Description
`type`	`Function` | keyof `HTMLElementTagNameMap`	Element type.
`props?`	`Record<string, any>`	Element properties.
`ref?`	`Key`	Element reference.

Returns `Element`

Array of bound calls.

Defined in packages/shell/src/component/runtime.ts:109

---

shell.createFragment

createFragment

▸ **createFragment**(`props?`): `Element`

JSX fragment factory. Provides a `jsx-runtime`-compliant helper function used by the transpiler to create JSX fragments.

Parameters

Name	Type	Description
`props?`	`Record<string, any>`	Fragment properties.

Returns `Element`

Array of bound calls.

Defined in packages/shell/src/component/runtime.ts:172

---

shell.customElements

customElements

• `Const` **customElements**: `CustomElementRegistry` & { `getName`: (`constructor`: `CustomElementConstructor`) => `undefined` | `string` }

Proxy around the built-in `customElements` object, maintaining a mapping of all registered elements and their corresponding names, which can be queried by calling `customElements.getName()`.

Remarks

https://github.com/WICG/webcomponents/issues/566

Defined in packages/shell/src/component/registry.ts:13

---

shell.references

references

▸ **references**(`target`): `Map<Key, Node> | undefined`

JSX reference helper. Calling this function while supplying a viable `target` will return all referenced JSX elements mapped by their corresponding keys known to the supplied `target`. A viable `target` may be any element, which previously was `target` to the render function.

Parameters

Name	Type	Description
`target`	`Element | DocumentFragment`	DOM element to resolve.

Returns `Map<Key, Node> | undefined`

Resolved references.

Defined in packages/shell/src/component/runtime.ts:196

---

shell.render

render

▸ **render**(`target`, `element`): `Node`

JSX rendering helper. This helper is a wrapper around the *patch* function from the *incremental-dom* library and renders a JSX `element` created through *createElement* into an `target` element or fragment.

See

- createElement
- patch

Parameters

Name	Type	Description
`target`	`Element | DocumentFragment`	Element or fragment to render into.
`element`	`Element`	JSX element to be rendered.

Returns `Node`

Rendered `target` element.

Defined in packages/shell/src/component/runtime.ts:214

---

shell.route

route

• `Const` **route**: typeof `route`

Symbol used as property key by the Route decorator to associate the supplied route configuration to the decorated component.

See

Route

Defined in packages/shell/src/router/route.ts:62

Module: state

state

• **state**: `Object`

`@sgrud/state` - The SGRUD state machine.

Defined in packages/state/index.ts:7

---

state.Global

Global

▸ **Global**(): (`constructor`: (`...args`: `any`[]) => `any`) => `void`

Returns `fn`

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	(`...args`: `any`[]) => `any`

Returns `void`

Defined in packages/state/src/store/global.ts:1

---

state.Local

Local

▸ **Local**(): (`constructor`: (`...args`: `any`[]) => `any`) => `void`

Returns `fn`

▸ (`constructor`): `void`

Parameters

Name	Type
`constructor`	(`...args`: `any`[]) => `any`

Returns `void`

Defined in packages/state/src/store/local.ts:1

---

state.StateHandler

StateHandler

• **StateHandler**: `Object`

Defined in packages/state/src/handler/handler.ts:12

---

state.StateHandler.constructor

constructor

• **new StateHandler**(`source?`, `thread?`)

Parameters

Name	Type
`source?`	`string`
`thread?`	`Thread<BusWorker>`

Defined in packages/state/src/handler/handler.ts:29

---

state.StateHandler.kernel

kernel

• `Private` `Readonly` **kernel**: `Kernel`

Defined in packages/state/src/handler/handler.ts:18

---

state.StateHandler.registerSource

registerSource

▸ `Private` **registerSource**(`source?`): `Observable<any>`

Parameters

Name	Type
`source`	`string`

Returns `Observable<any>`

.

Defined in packages/state/src/handler/handler.ts:48

---

state.StateHandler.registerTarget

registerTarget

▸ `Private` **registerTarget**(`port`): `Observable<any>`

Parameters

Name	Type
`port`	`MessagePort`

Returns `Observable<any>`

.

Defined in packages/state/src/handler/handler.ts:74

---

state.StateHandler.registerThread

registerThread

▸ Private **registerThread**(thread?): Observable<any>

Parameters

Name	Type	Default value
thread	Thread<BusWorker>	BusHandler.worker

Returns Observable<any>

.

Defined in packages/state/src/handler/handler.ts:96

---

state.StateHandler.serviceWorker

serviceWorker

• Private Optional **serviceWorker**: Remote<StateWorker>

Defined in packages/state/src/handler/handler.ts:23

---

state.Store

Store

• Abstract **Store**<S>: Object

Type parameters

Name	Type
S	extends Store = any

Defined in packages/state/src/store/store.ts:6

packages/state/src/store/store.ts:37

---

state.Store.[observable]

[observable]

• Readonly **[observable]**: () => Subscribable<State<S»

Type declaration ▸ (): Subscribable<State<S»

Returns Subscribable<State<S»

Defined in packages/state/src/store/store.ts:42

---

state.Store.constructor

constructor

• **new Store**<S>(state)

Type parameters

Name	Type
S	extends Store<any, S> = any

Parameters

Name	Type
state	State<S>

Defined in packages/state/src/store/store.ts:66

---

state.Store.dispatch

dispatch

▸ **dispatch**(...action): void

Parameters

Name	Type
...action	Action<S>

Returns void

Defined in packages/state/src/store/store.ts:75

---

state.Store.state

state

• get **state**(): State<S>

Returns State<S>

Defined in packages/state/src/store/store.ts:59

---

state.Store.changes

changes

• Private Readonly **changes**: BehaviorSubject<State<S»

Defined in packages/state/src/store/store.ts:47

---

state.Store

Store

• **Store**: Object

Defined in packages/state/src/store/store.ts:6

packages/state/src/store/store.ts:37

---

state.Store.Action

Action

T **Action**<`S`>: { [K in Exclude<keyof S, keyof Store>]: S[K] extends Function ? [K, ...P] : never }[`Exclude`<keyof `S`, keyof `Store`>]

Type parameters

Name	Type
`S`	extends `Store`

Defined in packages/state/src/store/store.ts:12

---

state.Store.State

State

T **State**<`S`>: { readonly [K in { [L in Exclude<keyof S, keyof Store>]: Required<S>[L] extends Function ? never : L }[Exclude<keyof S, keyof Store>]]: S[K] }

Type parameters

Name	Type
`S`	extends `Store`

Defined in packages/state/src/store/store.ts:22
